Specification and Analysis of Timing Constraints for Embedded Systems

Rajesh K. Gupta and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*— Embedded systems consist of interacting hardware and software components that must deliver a specific functionality under constraints on relative timing of their actions. We describe operation *delay* and execution *rate* constraints that are useful in the context of embedded systems. A delay constraint bounds the operation delay or specifies any of the thirteen possible constraints between the intervals of execution of a pair of operations. A rate constraint bounds the rate of execution of an operation and may be specified *relative* to the control flow in the system functionality. We present constraint propagation and analysis techniques to determine satisfaction of imposed constraints by a given system implementation. In contrast to previous purely analytical approaches on restricted models or statistical performance estimation based on runtime data, we present a static analysis in presence of conditionals and loops with the help of designer assists. The constraint analysis algorithms presented here have been implemented in a cosynthesis system, VULCAN, that allows the embedded system designer to interactively evaluate the effect of performance constraints on hardware-software implementation tradeoffs for a given functionality. We present examples to demonstrate the application and utility of the proposed techniques.**

***Index Terms*— Constraint analysis, constraint satisfiability, embedded systems, interactive analysis, rate constraints, relative rates, timing analysis.**

I. Introduction

THIS WORK considers a synthesis approach to the implementation of embedded systems under constraints on the timing performance of the system implementation and on the overall cost of design. An embedded system is targeted for a specific and limited application and, therefore, must be designed to efficiently implement the required functionality. Driven by the advances in semiconductor technology and computer-aided design techniques, embedded systems are increasingly used in new application areas such as automotive, networking, and consumer electronics. To address the complexity of the embedded system design task, recently there has been a surge of interest in use of *predesigned reprogrammable* components such as off-the-shelf microcontrollers to reduce the design time and design cost [1]–[4]. In *mixed* system implementations, portions of system functionality are realized by a program running on the processor. Consider for example, the design of a vehicular cruise controller described in Example 1.1 below.

Example 1.1: Fig. 1 shows a block diagram of the cruise controller. The controller monitors cruising speed, fuel consumption, and provides valve control and status/maintenance feedback to the driver. The clock, calibration, and get info portions are implemented in dedicated hardware, whereas the rest is implemented as a set of program routines *including a runtime system* running on a microprocessor. The controller performs velocity regulation by sampling the brake and gear inputs at least once every millisecond and delivering appropriate value to valve control at least once per second. In addition, the performance is also constrained by a maximum delay of 1 ms from the time a brake input is sampled to its effect on the valve control output.

A systematic exploration of system implementations using reprogrammable components requires specification and analysis of performance constraints to determine feasibility of an implementation in hardware or software. Related work on timing analysis comes from several sources such as feasible scheduling [5], [6] and rate analysis for asynchronous concurrent systems modeled using Petri nets [7]–[9]. More recently, Hulgaard *et al.* in [10] presented an exact algorithm for determination of bounds on the time interval between events in a process graph using implicit unfolding of the graph. In developing a tight bound on operation invocation intervals, it considers only process graphs without conditional invocation of operations. In contrast, this work considers timing analysis in presence of conditionals and loops. We present a constraint analysis procedure in which additional designer input can be specified to determine possible timing constraint violations. The use of designer input in resolving constraint violations is particularly valuable in presence of uncertainty due to conditional invocation of the operations. This paper describes (deterministic) constraint analysis as implemented in the VULCAN cosynthesis system. An overview of various cosynthesis subtasks for hardware and software including constraint analysis in the deterministic as well as in the probabilistic sense can be found in [11].

This paper is organized as follows. We briefly describe the input and its abstraction into a model in Section II. Section III presents specification of timing constraints and their abstraction into a constraint model. In Sections IV and V, we describe the notion of constraint satisfiability and present techniques to carry out constraint analysis. In Section VI, we describe

Manuscript received February 6, 1995; revised December 16, 1997. This paper was recommended by Associate Editor, R. Camposano. This work was supported in part by the AT&T Foundation, in part by NSF-ARPA MIP 9115432, and in part by NSF CAREER Award MIP 95-01615.

R. K. Gupta is with the Department of Information and Computer Science, University of California, Irvine, CA 92697 USA.

G. De Micheli is with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(97)04736-2.

0278–0070/97$10.00 © 1997 IEEE

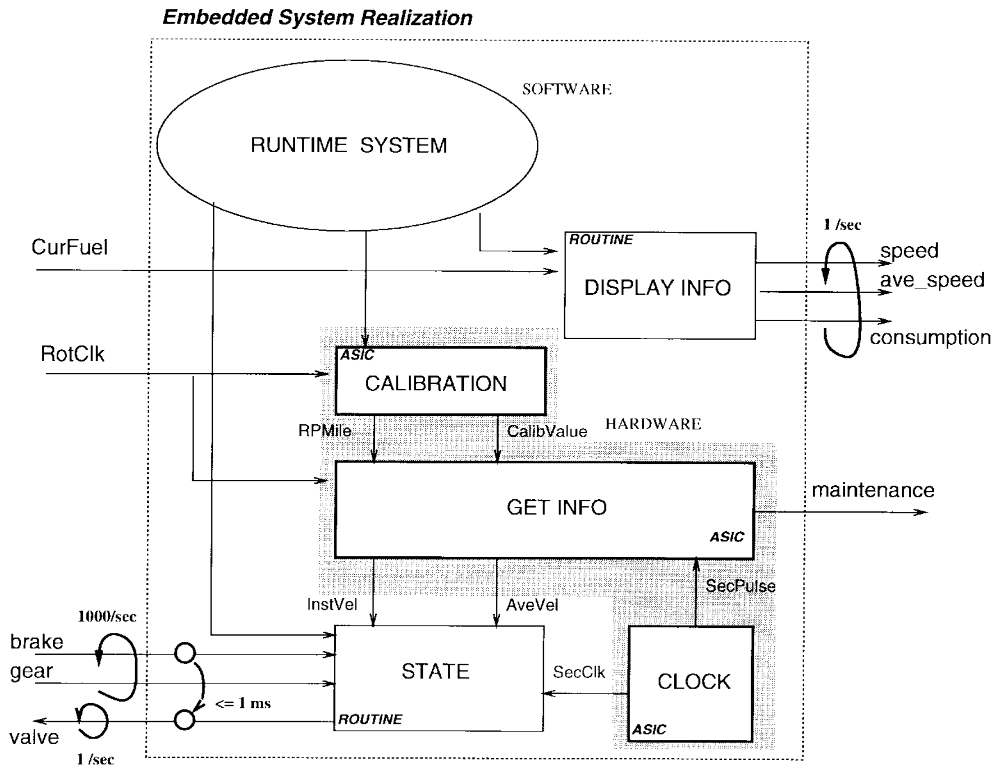

Fig. 1. Example of an embedded system implementation with timing constraints.

our implementation of the constraint analysis procedure and how it can be used by the system designer. We summarize contributions and discuss open issues in Section VII.

II. Input Specification and Model

We begin with a description of system functionality in a hardware description language (HDL), HardwareC [12]. Use of a HDL makes it possible to use existing synthesis techniques for digital hardware in system implementations consisting of both hardware and software components. Further, most HDL's allow for computation of explicit dependencies between operations and memory usage by use of static data types and unaliased data references. Both of these features are essential for analysis of constraints on timing and size of implementation. The particular choice of HardwareC is immaterial, and other procedural HDL's may be used as well.

The basic entity for specifying system behavior is a *process*. A process executes concurrently with other processes in the system specification. A process restarts itself on completion of the last operation in the process body. A process in HardwareC can have nested sequential and parallel statement blocks of statements. The use of multiple processes to describe a system functionality abstracts parts of a system implementation that operate at different speeds of execution. All communication between operations within a process body is based on shared memory. This shared storage is declared as a part of the process body. Interprocess communication is specified by message-passing operations that use a blocking protocol for synchronization purposes. This blocking is implemented using *wait* operations on associated control signals.

The input description is compiled into a graph based model, called flow graph, defined as follows.

Definition 2.1: A *flow graph model* is a polar acyclic graph, $G = (V, E, \chi)$, where $V = \{v_0, v_1, \cdots, v_N\}$ represent operations with v_0 and v_N being the source and sink operations, respectively. The edge set, $E = \{(v_i, v_j)\}$, represents dependencies between operation vertices. Function χ associates a Boolean (enabling) expression with every edge.

A vertex in the flow graph represents one of the following language-level operations: *nop, conditional, logic, arithmetic, io, wait*, and *link*. As mentioned earlier, the *wait* operation is used to represent synchronization events at model ports. The *link* operation is used to capture hierarchy of models by means of a call or a loop operation. The called flow graph corresponding to a link vertex may be invoked one or many times. Function and procedure calls are also represented by a call link vertex where the body of function/procedure is captured in a separate graph model.

A loop link operation consists of a loop condition operation that performs testing of the loop exit condition and a loop body. The loop body is represented as a separate graph model. The number of invocations of the loop body are controlled by a *loop index* variable associated with each loop operation. All loop operations are assumed to be of the form "repeat-until"; that is, a loop body is executed at least once. The HDL specification of "while"-loops is suitably transformed using a conditional operation. Any successor to a conditional operation is enabled if the result of condition evaluation selects the branch to which the operation belongs. This is expressed by the enabling condition associated with the edge leaving the

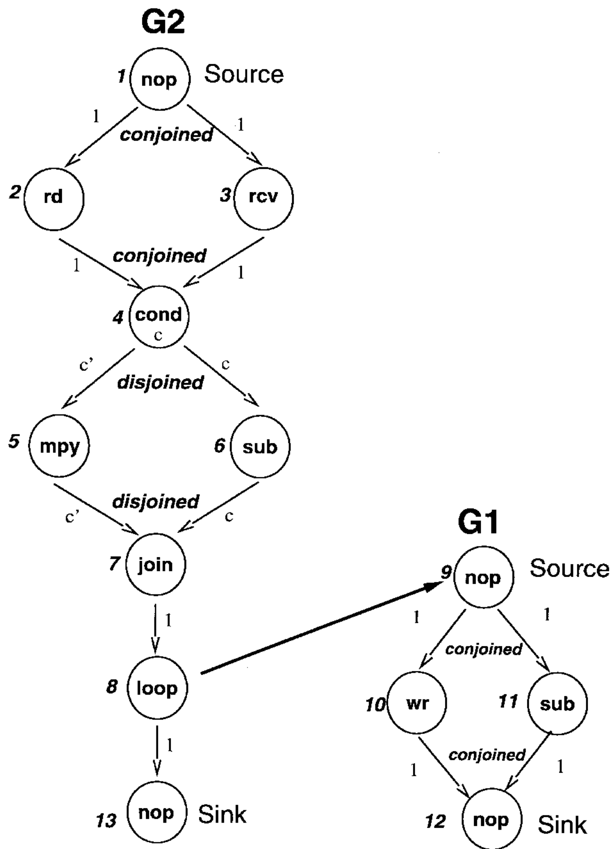

Fig. 2. An example of a flow graph model consisting of two graphs.

condition vertex. In general, a multiple in-degree operation vertex is enabled by evaluating an *input expression* consisting of logical AND and OR operations over enabling expressions of its fan-in edges. Similarly, on completion of an operation, all or a set of its successor vertices can be enabled. For each vertex, its *output expression* is an expression over enabling conditions of its fan-out edges. These expressions determine the flow of control through the graph model.

A flow graph is considered *well formed* if the input and output expressions use either AND or OR operations but not both in the same expression. For a well-formed graph, a set of input or output edges to a vertex is considered *conjoined* if the corresponding expression is a conjunction over inputs or outputs. Similarly, a set of edges is *disjoined* if the corresponding expression is a disjunction. Structurally, this makes the flow graph a *bilogic* graph [13]. For this reason, the flow graphs are called *bilogic sequencing graphs* as opposed to *unilogic* sequencing graphs in [12] which specify conditional paths as separate graphs. Bilogic graphs are a common occurrence in control graphs.

Example 2.1: Fig. 2 shows example of a well-formed bilogic graph model for a process consisting of a conditional and a loop operation. In subsequent examples, the labels "conjoined" and "disjoined" are also indicated by symbols "*" and "+," respectively.

Finally, a *system* consists of interacting parts, each of which can be abstracted into a flow graph model. A system model consists of one or more flow graphs that may be hierarchically linked to other flow graphs. That is, a system model is expressed as, $\Phi = \{G_1^*, G_2^*, \cdots, G_n^*\}$, where G_i^* represents the process graph model G_i and all the flow graphs that are hierarchically linked to G_i.

A. System Implementation

A flow graph model can be implemented as a hardware block or as a program. The implemented model is enabled by an external controller in case of hardware and by the runtime system in case of software, to perform the required tasks. Therefore, an *implementation* $\mathcal{I}(G)$ of a graph model G refers to assignment of delay and size properties to operations in G and a choice of *runtime scheduler* Υ that enable execution of the source operations in G. This actual assignment of values is related to the hardware or software implementation of operations in G. For nonpipelined hardware implementations, the runtime-scheduler is trivial, and the source operation is enabled once its sink operation has completed (and the graph enabling condition is true for conditionally invoked graphs). For software, the runtime scheduler refers to the choice of a runtime system that provides the operating environment for execution of operations in G. For details on implementation of the runtime system, the reader is referred to [14]. Note that a choice of operation schedule is not required for an implementation. This is because we assume that the expressed concurrency in flow graph models can be supported by available hardware resources. That is, any serialization required to meet hardware resource constraints has already been performed. This is not a strong assumption, since the availability of major hardware resources like adders and multipliers are usually known in advance.

Timing properties of a system model are derived using a bottom-up computation from individual operation delays. Let us first consider nonhierarchical flow graphs, i.e., graphs without link vertices. The *delay* δ of an operation refers to the execution delay of the operation in cycles. In a nonhierarchical flow graph, the delays of all operations (except *wait*) are fixed and independent of the input data. The *wait* operation offers variable delay which may or may not be data-dependent depending upon its implementation. The *latency* $\lambda(G)$ of a graph model G refers to the execution delay of G. The latency of a flow graph may be variable due to the presence of conditional paths. Next, the hierarchical flow graphs also contain link vertices, such as *call* and *loop* which point to flow graphs in the hierarchy. Therefore, an execution delay can be associated with link vertices as the latency of the corresponding graph model times the number of times the called graph is invoked. Since the latency can be variable, the delay of a link vertex can be variable. It may also be unbounded in case of loop vertices since these can, in principle, be invoked unbounded number of times.

The delay of *wait* operation depends upon its implementation. For instance, in a *busy-wait* implementation [i.e., *while(!signal)*], the wait operation is implemented as a loop operation that iterates until the concerned input *signal* is received. This implementation is commonly used for hardware synthesis [12]. Another implementation of wait operation is a

context-switch operation. The latter is particularly applicable for software implementations [15]. For this implementation, the delay of the wait operation is characterized as a fixed overhead as the delay due to the runtime system. This choice of operation delay is discussed later in Section V.

B. Path Lengths in Flow Graphs

We now describe the computation of path length parameters upon which our analysis of constraints is based. A path length $\ell(G)$ defines the lower bound on the latency of the longest path between the source and sink vertices, assuming the loop index to be unity for the loop operations.[1] In the presence of conditional paths, the length is a vector, $\underline{\ell} = (\ell[i])$, where each element $\ell[i]$ indicates the execution delay of a path in G. The elements of $\underline{\ell}$ are the lengths of the longest paths that are mutually-exclusive. Depending on system control flow, not all statically-derived paths may be executed. Therefore, some elements of $\underline{\ell}$ may well represent infeasible paths. It is possible to improve this estimation by providing user-directives as in [16], [17], though we do not consider such estimation methods here. However, the results of this analysis can be applied *mutatis mutandis* using dynamic path estimation methods to improve the constraint satisfiability tests presented later.

The length computation for a flow graph proceeds by a bottom-up computation of lengths from delays of individual operations. Given two operations, u and v with delays, δ_u, δ_v, these can be related in one of the following three ways in the flow graph.

- *Sequential composition*: The combined delay of u and v is the sum of two operation delays and is represented by $\delta_u \odot \delta_v \triangleq \delta_u + \delta_v$.
- *Conjoined composition*: When the operations u and v belong to two branches of a conjoined fork. Since the two operations are executed concurrently and the successor operation(s) can be started only after both u and v have completed execution, the combined delay of the two operations, in a well-formed bilogic flow graph, is defined by the maximum over the two operation delays $\delta_u \otimes \delta_v \triangleq \max(\delta_u, \delta_v)$.
- *Disjoined composition*: when the operations u and v belong to two branches of a disjoined fork. The delay of the composition is delay of u or delay of v depending upon the branch taken in an execution. Thus, the delay of the disjoined composition is a tuple representing possible operation delays $\delta_u \oplus \delta_v \triangleq (\delta_u, \delta_v)$.

As defined here, it can be shown that each of the three composition operators, $\odot$, $\otimes$, and $\oplus$ form a simple algebraic structure called commutative monoid, on the power set of positive integers Z^+ with zero as an identity element.

Operations $\oplus$ and $\otimes$ are easily extended to $k$ary operations. A disjoined composition of two delays leads to a two-tuple delay, since the two operations belong to mutually exclusive paths. This composition of delays is generalized to composition of paths as follows. In the case of a sequential composition of two path lengths, $\underline{\ell}_u$ and $\underline{\ell}_v$ with cardinality n and m respectively, the resulting path length contains $n \times m$ elements, consisting of sums over all possible pairs of elements of $\underline{\ell}_u$ and $\underline{\ell}_v$. In the case of a conjoined composition, the resulting path length is of cardinality $n \times m$ and consists of maximum over all possible pairs of elements. Finally, in a disjoined composition, the resulting path length is of cardinality $n + m$ and contains all elements of $\underline{\ell}_u$ and $\underline{\ell}_v$.

[1] Recall that loop vertices represent "repeat-until" type operations. The length computation treats the loop operation as a call operation.

In practice, one often needs only the upper and lower bounds on latencies. Notationally, ℓ_m and ℓ_M refer to the minimum and maximum element in $\underline{\ell}$, respectively. For well-formed graphs, ℓ_m and ℓ_M can be computed efficiently by collapsing conditional paths into a single operation vertex with minimum or maximum branch delay respectively. We state without proof the following properties:

$$(\underline{\ell}_1 \odot \underline{\ell}_2)_M = \ell_{1M} + \ell_{2M} \quad (1)$$
$$(\underline{\ell}_1 \odot \underline{\ell}_2)_m = \ell_{1m} + \ell_{2m} \quad (2)$$
$$(\underline{\ell}_1 \otimes \underline{\ell}_2)_M = \max(\ell_{1M}, \ell_{2M}) \quad (3)$$
$$(\underline{\ell}_1 \otimes \underline{\ell}_2)_m = \max(\ell_{1m}, \ell_{2m}) \quad (4)$$
$$(\underline{\ell}_1 \oplus \underline{\ell}_2)_M = \max(\ell_{1M}, \ell_{2M}) \quad (5)$$
$$(\underline{\ell}_1 \oplus \underline{\ell}_2)_m = \min(\ell_{1m}, \ell_{2m}). \quad (6)$$

These properties have been used in previous work related to execution path analysis in [16]. Note that our flow graphs are derived from descriptions in a structured programming language. In other words, in the input descriptions, there are no jumps and goto statements. This makes the control flow in the flow graphs series-parallel where the blocks (groups of operation vertices) are either nested or sequential. This simplifies the path length computation as the following example shows.

Example 2.2—Latency and Path Length Computations for Bilogic Flow Graphs: Fig. 3 below shows a process graph model G_3 and graph models on its calling hierarchy. G_3 calls G_2 that constitutes body of a loop operation v_3. G_2, in turn, calls G_1 that constitutes the body of a loop operation v_2. Numbers in the circle indicate delay of the operations. For this set of graph models, the path lengths are

$$\begin{aligned}
\underline{\ell}(G_1) &= 2 \odot (0,1) \odot (1) = (3,4)\\
\underline{\ell}(G_2) &= 2 \odot 3 \odot (1,6) \odot \underline{\ell}(G_1)\\
&= (6,11) \odot \underline{\ell}(G_1) = (9,10,14,15)\\
\underline{\ell}(G_3) &= 2 \odot 2 \odot (0,(5,7)) \odot \underline{\ell}(G_2)\\
&= 4 \odot (0,5,7) \odot \underline{\ell}(G_2) = (4,9,11) \odot \underline{\ell}(G_2)\\
&= (13,14,18,19,20,21,23,24,25,26).
\end{aligned}$$

C. Rate of Execution

The *instantaneous rate of execution* $\tilde{\rho}_i(t)$ of an operation v_i is the marginal number of executions n of operation v_i at any instant of time, t, $\tilde{\rho}_i(t) \triangleq \frac{dn}{dt} = \lim_{\Delta t \to 0} \frac{\Delta n}{\Delta t}$ per second. Due to the discrete nature of executions (i.e., $n \in Z^+$), we define

$$\tilde{\rho}_i(t) \triangleq \begin{cases} \frac{1}{t - t_k(v_i)} & k \text{ such that } t_k(v_i) < t < t_{k+1}(v_i) \\ 0 & t \le t_1(v_i) \end{cases}$$

where $t_k(v_i)$ refers to the start time of the kth execution of operation v_i. Assuming a synchronous execution model with cycle time τ, we define the (lattice) rate of execution

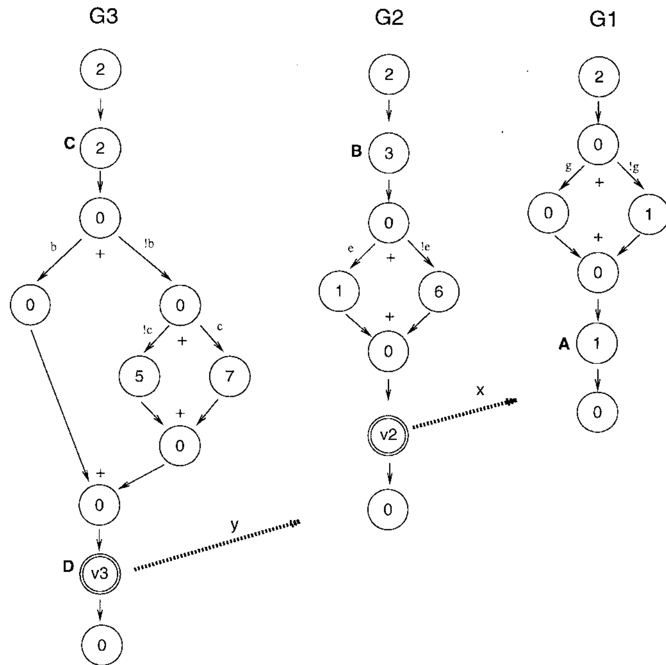

Fig. 3. Path length computations.

at invocation k of an operation v_i as the inverse of the time interval between its current and previous execution. That is

$$\rho_i(k) \triangleq \tilde{\rho}_i(t)\mid_{t=t_k(v_i)} = \frac{1}{t_k(v_i) - t_{k-1}(v_i)}(\text{sec}^{-1}) = \frac{\tau}{t_k(v_i) - t_{k-1}(v_i)}(\text{cycle}^{-1}). \quad (7)$$

By convention, the instantaneous rate of execution is zero at the first execution of an operation ($t_0 \rightarrow -\infty$). Note that ρ_i is defined only at times when operation v_i is executed, whereas $\tilde{\rho}_i$ is a function of time and defined at all times. In our treatment of execution rates and constraints on rates, only rates at times of operation execution are of interest. Hence, we use the definition of ρ as the rate of execution.

For a graph model, G, its *rate of reaction*, is the rate of execution of its source operation, that is, $\varrho_k(G) \triangleq \rho_0(k)$. The reaction rate is a property of the graph model, and it is used to capture the effect on the runtime system and the type of implementation chosen for the graph model. To be specific, the choice of a nonpipelined implementation of G leads to $\varrho_k(G)^{-1} = \lambda_k(G) + \gamma_k(G)$ where $\gamma_k(G)$ refers to the *overhead delay*, that represents the delay in reinvocation of G. $\gamma_k(G)$ may be a fixed delay representing the overhead of a runtime scheduler or it may be a variable quantity representing delay in case of conditional invocation of G.

D. Nondeterminism in Flow Graph Models

The delay of a basic operations in the HDL input can be characterized by a fixed number related to its implementation in hardware or software. However, the delay of some operation vertices in flow graph model depends on the input data, specifically either on the *value* of input data or on the *timing* of input data. An example of an operation with value-dependent delay is loop link operation with a data-dependent loop index. Since the execution delay (or latency) of a bilogic flow graph can, in general, be data-dependent due to the presence of conditional paths, the delay of a call vertex is also variable and data-dependent. Therefore, in bilogic flow graphs, link vertices present value-dependent delays. An operation presents a input timing-dependent delay only if it has *blocking* semantics. The only operation in the flow graph model with blocking semantics is the *wait* operation. (Note that the *read* and *write* operations are treated as nonblocking.) Data-dependent loop and synchronization operations introduce uncertainty over the precise delay and order of operations in the system model. Due to concurrently executing flow graph models, these operations affect the order in which various operations are invoked. Due to this uncertainty, a system model containing these operations is called a *nondeterministic* [18] model, and operations with variable delays are termed *nondeterministic delay* or $\mathcal{ND}$ operations.

TABLE I
POSSIBLE TIMING CONSTRAINTS BETWEEN A PAIR OF OPERATIONS

Constraint type	*Description*
A before B	$t_a + d_a < t_b$
A meets B	$t_a + d_a = t_b$
A overlaps B	$t_b - t_a < d_a$
A finishes by B	$t_a + d_a = t_b + d_b$
A during B	$t_a > t_b$ and $t_a + d_a < t_b + d_b$
A finishes B	$t_a > t_b$ and $t_a + d_a = t_b + d_b$
A is overlapped by B	$t_a - t_b < d_b$
A is met by B	$t_a + d_a = t_b$
A after B	$t_b + d_b < t_a$
A contains B	$t_a < t_b$ and $d_a > d_b$
A starts B	$t_a = t_b$ and $d_a < d_b$
A equals B	$t_a = t_b$ and $d_a = d_b$
A is started by B	$t_a = t_b$ and $d_a > d_b$

III. TIMING CONSTRAINTS

Operation-level timing constraints are of two types:

a) operation *delay* constraints;
b) execution *rate* constraints.

Delay constraints are either unary such as bounds on the delay of an operation or binary as bounds on the delay between initiation time of two operations. Given any pair of operations A and B, different possible constraints on the interval of execution of the two operations are described in Table I. Here, symbols t_a and d_a represent execution start time and execution delay of operation a. We define *minimum timing constraint* $l_{ij} \geq 0$ from operation vertex v_i to v_j as

$$t_k(v_j) \geq t_k(v_i) + l_{ij}. \quad (8)$$

By default, any sequencing dependency between two operations, induces a minimum delay constraint which must be satisfied in order to observe the execution semantics of the

flow graph. Similarly, a *maximum timing constraint*, $u_{ij} \geq 0$ from v_i to v_j, is defined by the following inequality:

$$t_k(v_j) \leq t_k(v_i) + u_{ij}. \tag{9}$$

We note that the operation delay constraints are general and can be used to capture durational and deadline constraints in specifying real-time systems [19]. For a given operation v, release time $R(v)$ is indicated by a minute delay constraint of $R(v)$ from source to v. Similarly, a deadline of $D(v)$ is indicated by a max delay constraint of $D(v) - d(v)$ from v to source operation where $d(v)$ is the execution delay of the operation v.

Execution rate constraints refer to bounds on the rate of execution of an operation. In particular, execution rate constraints on input (output) operations refer to the rates at which the data is required to be consumed (produced). We assume that each execution of an input (output) operation consumes (produces) a *sample* of data. Execution rate constraints on I/O operations are referred to as *data rate constraints*. A *minimum data rate constraint*, r_i (samples per cycle or cycles^{-1}) on an I/O operation defines the lower bound on the execution rate of operation v_i. Similarly, a *maximum data rate constraint* R_i (cycles^{-1}) on an I/O operation defines the upper bound on the execution rate of operation v_i. That is

$$\begin{aligned} &\rho_{v_i}(k) \leq R_i, && \text{for all } k>0 \quad \text{[max rate]} \\ \Rightarrow \quad &t_k(v_i) - t_{k-1}(v_i) \geq \tau \cdot {R_i}^{-1}, && \text{for all } k>0. \end{aligned} \tag{10}$$

Similarly

$$\begin{aligned} &\rho_{v_i}(k) \geq r_i, && \text{for all } k>0 \quad \text{[min rate]} \\ \Rightarrow \quad &t_k(v_i) - t_{k-1}(v_i) \leq \tau \cdot {r_i}^{-1}, && \text{for all } k>0. \end{aligned} \tag{11}$$

In embedded system designs, a rate constraint is often specified relative to a specific mode of operation, for example, sampling of gear control only when the car is moving in case of an automotive controller. These constraints are captured using relative rate constraints described next.

A. Relative Rate of Execution

In general, when considering the rate of execution of v_i, we must consider the successive executions of v_i that may belong to separate invocations of G. A *relative execution rate constraint* of an operation v_i, with respect to a graph model G, is a constraint on the rate of execution of v_i with respect to the same invocation of G, that is, *when G is continuously enabled and executing*. In other words

$$r_i^G \leq \rho_{v_i}(k) \leq R_i^G \tag{12}$$

for all $k > 0$, *and* there exists an execution j of G such that

$$t_j(v_0(G)) \leq t_{k-1}(v_i) \leq t_k(v_i) \leq t_j(v_N(G))$$

where v_0 and v_N refer to source and sink vertices in G, respectively. The relative rate of execution expresses rate constraints that are applicable to a specific *context* of execution as expressed by the control flow in G. Clearly, a relative rate constraint is meaningful when expressed relative to a flow graph in the hierarchy in which the operation resides.

B. Specification of Timing Constraints

Operations in the flow graph model correspond to language-level operations, that is, operations supported in the HDL. Therefore, it is easy to specify timing constraints by tagging the corresponding statements in HDL descriptions. In the case of nested loop operations, rate constraints are indexed by the corresponding loop operations. The loops are indexed by increasing integers where the inner-most loop is indexed zero. In Example 3.1 below, there are two relative rate constraints on the *read* operation with respect to the two *while* statements.

Example 3.1: The following example, shown at the bottom of the next page and derived from the transmit process of an ethernet controller, demonstrates the specification of rate constraints in presence of nested loop operations. Recall that each execution of read operation indicated by tag "A" produces a sample of data. The rate constraints attributes use inverse of the rates in units of "cycles/sample." In this example, a minimum rate constraint of 100 cycles per sample execution, or 0.01 executions per cycle is specified on the read operation. In addition, two *relative* minimum rate constraints of one and 0.1 per cycle are specified for the read operation relative to the loops *while(bitEN)* and *while(frameEN)*, respectively.

Operation delay constraints are specified similarly using the following syntax:

- constraint mintime from $\langle$tag1$\rangle$ to $\langle$tag2$\rangle$ = $\langle$num$\rangle$ cycles;
- constraint maxtime from $\langle$tag1$\rangle$ to $\langle$tag2$\rangle$ = $\langle$num$\rangle$ cycles;
- constraint finish—before—during $\langle$tag1$\rangle$ $\langle$tag2$\rangle$;

□

IV. Relationship of Constraint Analysis to Operation Scheduling

For each invocation of a flow graph model, an operation is invoked zero, one or many times depending upon its position on the hierarchy of the flow graph model. The execution times of an operation are determined by the two separate mechanisms:

a) the runtime scheduler, Υ;
b) the operation scheduler, Ω.

The runtime scheduler determines the invocation times of flow graphs, which may be as simple as fixed-ordered where the selection is made by a predefined order (most likely by the system control flow). This is typically the case in hardware implementations where the graph invocation is determined entirely by the system control flow. Software implementations of the runtime scheduler are based on the choice of the runtime environment which is characterized by a *fixed* delay overhead for each runtime operation.

Given a graph model, $G = (V, E)$, the selection of a *schedule* refers to the choice of a function Ω that determines the start time of the operations such that

$$t_k(v_i) \geq \max_{j \ni v_j < v_i} [t_k(v_j) + \delta(v_j)] \tag{13}$$

is satisfied for each invocation $k > 0$ of operations v_i and v_j. Here $\delta(\cdot)$ refers to the delay function and returns the execution delay of the operation. $v_j < v_i$ indicates the dependency of operation i on operation j.

Given a scheduling function, a timing constraint is considered *satisfied* if the operation initiation times determined by the scheduling function satisfy the corresponding inequalities [see (8)–(11)]. Clearly, the satisfaction of timing constraints is related to the choice of the scheduling function. In general, the choice of a particular operation scheduling mechanism depends upon the types of operations supported and the resulting control hardware or software required to implement the scheduler. We determine constraint satisfiability in the context of a bilogic relative scheduler used in our synthesis system. For presentation purposes, we briefly review the relative scheduling mechanism for unilogic flow graphs, followed by a straightforward extension to the bilogic flow graphs.

We consider first a model G where the delay of all operations in G is known and bounded. A schedule of G maps vertices to integer labels that define the start time of corresponding operations, that is, $\Omega_s : V \mapsto Z^+$ such that operation start times, $t_k(v_i) = \Omega_s(v_i)$ satisfy inequality (13). A schedule is considered minimum if $|t_k(v_i) - t_k(v_o)|$ is minimum for all $v_i \in V$. (Recall that we assume that the effect of resource constraints has been taken into account as additional serializations in the graph model.) For each invocation of G, since the start times of all operations are fixed for all executions of G (that is, for all k), such a schedule is referred to as a *static schedule*.

In general, due to the conditional, loop, and wait operations, not all delays can be fixed or known statically, thus making a determination of an unique operation start time impossible for a static scheduler. This problem is addressed by a relative scheduler [6] that uses runtime information to determine operation start times for each invocation of a graph model. As a result, a relative scheduler does not require $\delta(\cdot)$ to be a fixed quantity. A *relative schedule* function maps vertices to a *set* of integers representing *offsets*. An offset $\theta_{v_j}(v_i)$ of vertex v_i with respect to vertex v_j is defined as the delay in starting execution of v_i after completion of operation v_j. Offsets are determined relative to vertices which the execution of v_i (transitively) depends upon. That is

$$t_k(v_i) \geq t_k(v_j) + \delta(v_j) + \theta_{v_j}(v_i), \quad \text{if } v_j <^* v_i$$

where $<^*$ represents transitive closure of the dependency relation $<$. For a given vertex v_i, a set $\mathcal{A}(v_i)$ of *anchor* vertices is defined as the set of conditional $(\mathcal{CD})$ and loop wait $(\mathcal{ND})$ vertices that have a path to v_i

$$\mathcal{A}(v_i) = \{v_j \in V : v_j <^* v_i, v_j \text{ is } \mathcal{ND} \text{ or } \mathcal{CD}\}. \tag{14}$$

A relative schedule function Ω_r is defined as a set of offsets for each operation such that operation start time satisfies the inequality

$$t_k(v_i) \geq \max_{a \in \mathcal{A}(v_i)} [t_k(a) + \delta(a) + \theta_a(v_i)]. \tag{15}$$

Since the quantity $\delta(a)$ is known only at runtime, the operation start time under a relative schedule is determined only at the runtime.

Inequality (15) can be derived from the inequality (13) by expressing the latter over the transitive closure $G^{<^*}$ of G and then adding the known operation delays δ as offsets from unknown delay operations. Clearly, a solution to inequality (15) will also satisfy inequality (13) if the offsets, $\theta_{v_j}(v_i) \geq \ell(v_j, v_i)$, where $\ell(v_j, v_i)$ refers to the path length from vertex v_j to vertex v_i. Finally, a relative schedule is considered minimum if it leads to minimum values of all offsets for all vertices.

One of the interesting properties of a relative schedule is that it expresses the (spatial) uncertainty associated with conditional invocations of an operation $(\mathcal{CD})$ as its temporal uncertainty by treating it as an unbounded delay $(\mathcal{ND})$ operation. Thus, a conditional operation is same as a data-dependent loop operation where operations on its branches are invoked a variable number of times (zero or one) depending upon data values. For the purposes of relative scheduling, variable delay operations are treated as unknown delay operations in [20]. Due to this treatment, the corresponding flow graph used

```
process example (frameEN, bitEN, bit, word)
      in port frameEN, bitEN, bit;
      out port word[8];
{
      boolean store[8], temp;
      tag A;
      while (frameEN)
      {
            while (bitEN)
            {
A:                temp = read(bit);
                  store[7:0] = store[6:0] @ temp;
            }
            write word = store;
      }
      attribute "constraint minrate of A = 100 cycles/sample";
      attribute "constraint minrate 0 of A = 1 cycles/sample";
      attribute "constraint minrate 1 of A = 10 cycles/sample";
}
```

for relative scheduling is *unilogic*, since conditional branches belong to separate graphs same as in the case of loops. Of course, this idea can be carried further by treating all operations as unbounded delay operations and computing the start times of operations at runtime. Such an implementation of a flow graph would be similar in architecture to data flow machines [21]. In terms of the latency of execution, such a dynamic scheduler will give the most "compact" schedule. However, the increased overhead cost of the runtime system associated with a large number of unbounded delay operations would make such an architecture unsuitable for either gate-level hardware or constrained software implementations. Hence, attempts have been made to minimize the number of unknown delay operations in the graph model that belong to an anchor set [22]. The relative scheduler is extended to bilogic relative scheduler by treating conditionals separately from loop and synchronization operations. In particular, the maximum path delay over all branches in a conditional is used for scheduling purposes. Depending upon the actual branch taken, this schedule may not be the minimum in the sense of relative scheduling described earlier. However, it reduces the number of $\mathcal{ND}$ operations, thus making it easier to perform the constraint analysis. Also, the cost of implementing control for a bilogic relative scheduler lies somewhere between the control costs for static and relative schedulers.

A *bilogic relative schedule* treats an operation offset as a *vector* $\underline{\theta}_{v_j}(v_i)$ representing the (finite) set of possible delays. A bilogic schedule, Ω_{br} then computes the offset vectors such that

$$t(v_i) \geq \max_{a \in \mathcal{A}_b(v_i)} [t_a + \delta(a) + |\underline{\theta}_a(v_i)|_\infty] \tag{16}$$

where $|\cdot|_\infty$ refers to the largest element (or the infinity norm) of the vector. The bilogic anchor set is defined as $\mathcal{A}_b(v_i) = \{v_j \in V : v_j <^* v_i, v_j \text{ is } \mathcal{ND}\}$. Once again, the inequality (16) can be derived from inequality (13) for bilogic flow graphs. Thus, a solution to inequality (16) will also satisfy inequality (13) provided $|\underline{\theta}_a(v_i)|_\infty \geq \ell_M(a, v_i)$. The following shows an example of unilogic and bilogic relative schedules.

Example 4.1: Let us consider the unilogic versus bilogic relative schedules for the flow graph model shown in Example 2.1. We assume that the operation mpy takes three cycles while the rest take one cycle. The assignment of offsets using a relative scheduler and a bilogic relative scheduler are shown in Table II, where a "–" indicates that the start time of the operation is not affected by the particular anchor vertex.

For example, consider the *relative schedule* of vertex v_7. Its offset vector is {1, 0, 0, -}. This indicates that v_7 starts after one cycle from v_1 and immediately after start of v_3 and v_4. Also, the start time of this vertex is not affected by the start time of v_8. Therefore, according to inequality (15), the start time of v_7 is given by

$$t(v_7) = \max\{t(v_1) + 1, t(v_3) + \delta_3, t(v_4) + \delta_4\}.$$

Note that $\delta(v_1) = 0$. For a *bilogic relative scheduler*, the vertex v_3 is no longer an anchor but a variable delay vertex. The offsets are now computed as vectors of possible delay values. Thus, the start time for vertex, v_7 in this case is given by

$$t(v_7) = \max\{t(v_1) + 4, t(v_3) + \delta_3 + 3\}.$$

TABLE II

Vertex	Relative offset, θ				Bilogic relative offset, $\underline{\theta}$		
	v_1	v_3	v_4	v_8	v_1	v_3	v_8
1	-	-	-	-	-	-	-
2	0	-	-	-	0	-	-
3	0	-	-	-	0	-	-
4	1	0	-	-	1	0	-
5	-	-	0	-	1	0	-
6	-	-	0	-	1	0	-
7	1	0	0	-	(2,4)	(1,3)	-
8	1	0	0	-	(2,4)	(1,3)	-
9	1	0	0	-	(2,4)	(1,3)	-
10	-	-	-	0	-	-	0
11	-	-	-	0	-	-	0
12	-	-	-	1	-	-	1
13	1	0	0	0	(2,4)	(1,3)	0

A. Constraint Satisfiability

The satisfiability of a given set of constraints requires that there exist an implementation that satisfies the imposed constraints. For constraint analysis purposes, it is not necessary to determine a schedule of operations, but only to verify the *existence* of a schedule. Since there can be many possible schedules, constraint satisfiability analysis proceeds by identifying conditions under which *no* solutions are possible. A timing constraint is considered *inconsistent* if it can not be satisfied by *any* implementation of the flow-graph model. A set of timing constraints is considered *mutually inconsistent* if these constraints can not be satisfied by any implementation of the flow graph model. Since the consistency of constraints is independent of the implementation, these are related to the structure of the flow graphs.

The timing constraints are abstracted in a constraint graph model consisting of vertices as operations, forward edges indicating minimum delay constraints, and backward edges indicating maximum delay constraints.

Definition 4.1: The *timing constraint graph model* G_T is defined as $G_T = (V, E, \Delta)$, where the set of edges consists of *forward* and *backward* edges, and $E = E_f \cup E_b$ and $\delta_{ij} \in \Delta$ define the weights on edges such that $t_k(v_i) + \delta_{ij} \leq t_k(v_j)$ for all $k > 0$.

Deterministic analysis refers to constraint satisfiability determination over all execution instances. This analysis relies on the information that is statically available, such as operations and dependencies. The static information is captured very well by the constraint graphs. However, as we show later, constraint satisfiability based solely on static information may be impossible, and additional user information may be needed

to provide deterministic answers to constraint satisfiability. We characterize those cases where additional user input is needed to proceed with constraint analysis. The rationale and use of additional information is explained later in Section V. The key contribution of the deterministic analysis is that not all constraints require bounding of all operation delays. Therefore, static path analysis provides a valuable input to the system designer in making a choice of system components under timing constraints.

For consistent timing constraints, the analysis is performed in stages and in order of increasing nondeterminism in the model. We first consider the satisfiability of operation delay constraints followed by the execution rate constraints.

V. Constraint Satisfiability Tests

There are two important previous results that lay down the conditions for determining constraint satisfiability. The following theorem occurs in various forms in different application areas. Its proof can be found in, for example, in [5], [20], and [23].

Theorem 5.1—Static Scheduling: In the absence of any $\mathcal{ND}$ operations, a set of operation delay constraints is satisfiable, if and only if, there exist no positive cycles in G_T.

Using a relative scheduler, a minimum delay constraint is always satisfiable since from any solution that satisfies inequalities (15) or (16), a solution can be constructed such that $\theta_{v_j}(v_i) \geq \max(\ell(v_j, v_i), l_{ji})$ for each constraint l_{ji}. This solution satisfies both inequalities (13) and (8). On the contrary, a maximum delay constraint may not always be satisfiable. A constraint graph is considered *feasible* if it contains no positive cycle when the delay of $\mathcal{ND}$ operations is assigned to zero. The following theorem [6] lays out a necessary and sufficient condition to determine the satisfiability of constraints in presence of $\mathcal{ND}$ operations.

Theorem 5.2—Relative Scheduling: Min/max delay constraints are satisfiable if and only if the constraint graph is feasible and there exist no cycles with $\mathcal{ND}$ operations.

Execution rate constraints are constraints on the time interval between invocations of the same operation. In general, this interval can be affected by pipelining techniques. We consider here only nonpipelined implementations of the flow-graph models. Therefore, operations in the graph model are enabled for the next iteration only after completion of the previous iteration

$$t_{k-1}(v_0) \leq t_{k-1}(v_N) \leq t_k(v_0) \leq t_k(v_N), \quad \text{for all } k > 0. \tag{17}$$

Consider an I/O operation $v_i \in V(G)$ with data-rate constraints, r_i and R_i. The rate constraints imply

$$\frac{\tau}{R_i} \leq t_k(v_i) - t_{k-1}(v_i) \leq \frac{\tau}{r_i}, \quad \text{for all } k > 0. \tag{18}$$

τ refers to the cycle time of the clock associated with G. Inequality (18) is satisfied if and only if

$$\min_k(t_k(v_i) - t_{k-1}(v_i)) \geq \frac{\tau}{R_i} \quad \text{[lower bound]} \tag{19}$$

$$\max_k(t_k(v_i) - t_{k-1}(v_i)) \leq \frac{\tau}{r_i} \quad \text{[upper bound].} \tag{20}$$

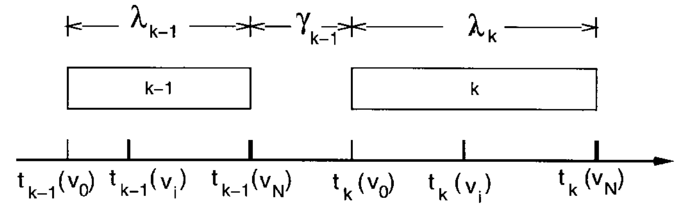

Fig. 4. Operation invocation interval.

Thus, satisfiability for execution rate constraints is determined by checking for the minimum and maximum delay between any two consecutive invocations of constrained operation. This interval can be expressed as (see Fig. 4)

$$\begin{aligned} &t_k(v_i) - t_{k-1}(v_i) \\ &\quad= [t_k(v_i) - t_k(v_0)] + [t_k(v_0) - t_{k-1}(v_N)] \\ &\qquad+ [t_{k-1}(v_N) - t_{k-1}(v_0)] + [t_{k-1}(v_0) - t_{k-1}(v_i)] \\ &\quad= \lambda_k(v_i) + \gamma_{k-1}(G) + \lambda_{k-1}(G) - \lambda_{k-1}(v_i) \end{aligned} \tag{21}$$

where $\lambda_k(v_i)$ refers to execution delay from source vertex v_0 to v_i for the kth execution. $\gamma_{k-1}(G)$ is the delay in rescheduling a graph, that is, the time from completion of $(k-1)$th execution of G to initiation of the kth execution. From inequalities (13) and (17) each of the four components in inequality (21) are nonnegative quantities. Let us now consider the lower and upper bounds on this interval. These bounds are developed based on the analysis of paths in the flow graph. It follows from inequality (13) that for vertices in a path, $p = \{v_i, v_{i+1}, \cdots, v_j\}$ the following is true for all $k > 0$

$$t_k(v_i) \leq t_k(v_{i+1}) \leq \cdots \leq t_k(v_j). \tag{22}$$

It is important to note that even though the actual interval between successive executions is summed as shown in (21), the bounds on this interval can be developed based on analysis of the graph model itself. This is because, in a nonpipelined implementation of G, the consecutive execution of an operation corresponds to traversal of a path from source to sink vertex in G. Consider $(k-1)$th and kth executions of an operation v_i in $V(G)$ as shown in Fig. 5. Let $q_{k-1} = \{v_i, \cdots, v_N\}$ represent the path traversed from v_i to v_N in $k-1$th execution of G and let $p_k = \{v_0, \cdots, v_i\}$ be the path traversed from v_0 to v_i in kth execution of G. Using inequality (22) it can be easily shown that $p_k \cup q_{k-1}$ is a path from source to sink in G.

Theorem 5.3—Maximum Rate Constraint: A max-rate constraint R_i in G is satisfied if $\ell_m(G) \geq \tau \cdot R_i^{-1}$.

Proof: In order to obtain a lower bound on the interval between two consecutive executions of operation v_i, we consider the case when the execution of the graph model is restarted immediately after the completion of the previous execution, i.e., $\gamma_{k-1}(G) = 0$. From the discussion above, there exists a path in G that corresponds to the consecutive execution of operation v_i. In other words, the interval $t_k(v_i) - t_{k-1}(v_i)$ is bounded by the latency of the graph. Recall that the length vector provides a lower bound on latency of G. The result follows. □

Note that similar to a minimum delay constraints, a maximum rate constraint is always satisfiable. When $\ell_m(G) <$

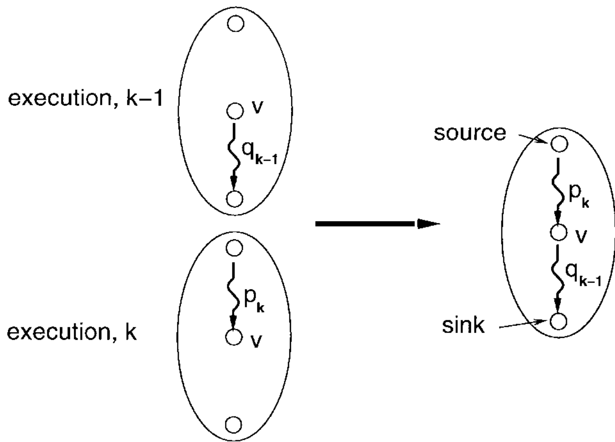

Fig. 5. Consecutive executions of an operation corresponds to traversal of a path in G.

$\tau \cdot R_i^{-1}$, the maximum rate constraint R_i can still be satisfied by an appropriate choice of the overhead delay that is applied to every execution of G.

Example 5.1—Maximum Rate Constraints: For the flow graph model shown in Fig. 2, the maximum rate of the *write* operation, determined by $\ell(G_1)$, is 1 cycle^{-1}, whereas the maximum rate of the *read* operation is determined by $\underline{\ell}(G_2) = ((1 \otimes 0) \odot (1 \oplus 3)) \odot (1 \otimes 1) = (3, 5)$ is $1/3$ cycle^{-1}. Any maximum rate constraint larger than or equal to $1/3$ is satisfied by the graph model.

Note that this lower bound ℓ_m, used for checking the satisfaction of maximum rate constraints, also defines the fastest rate at which an operation in the graph model can be executed by a nonpipelined implementation. This points to the necessary condition for meeting a minimum rate constraint. Sufficient conditions for minimum rate constraints are considered next.

A. Constraint Satisfiability Tests Using Graph Hierarchy

A hierarchical flow graph is composed using link vertices. A link vertex represents a call to a flow graph in the hierarchy. For a given graph model G, that is called by a link vertex in graph G', graph G' is also referred to as a parent graph of G and is considered to be *above* G in the control-flow hierarchy G in the system control-flow. While the lower bound on the time-interval between successive executions of an operation can be derived by analyzing G^*, that is the graph to which the operation belongs, and all the graphs *below* in the control-flow hierarchy, the determination of the upper-bound on the interiteration interval of an operation, also requires estimations of the delays due to operations and graphs that lie *above* the operation in the control-flow hierarchy. In particular, the effect of the runtime scheduler must also be taken into account.

We use the following notation to help express the propagation of constraints over the graph hierarchy. For a given graph G, G_+ denotes the parent body that calls the graph G. For a graph G, G_o refers to the *parent process graph*, i.e., the graph at the root of the hierarchy corresponding to a process model.

Note that (static) determination of the interval of successive executions of an operation that is conditionally invoked is undecidable. That is, there may not exist an upper bound on the invocation interval. For example, consider a statement *if (condition) value = read (a)*;. There is not enough information to determine the rate of execution of the *read* operation. For deterministic analysis purposes, we take a two step approach to answering constraint satisfiability.

1) Determine if a given implementation is satisfactory *assuming that the condition is always true*. In other words, the only uncertainty is conditional invocation of the graph which may correspond to the body of a process or a loop operation. This is consistent with the interpretation that a timing constraint specifies a bound on the interval between operation executions but does not imply, *per se*, that the operation must be executed. Under this assumption, the loops are executed at least once (that is, loops are of the type "repeat-until"), since the "while" loops are expressed as a conditional, followed by a repeat-until loop, as explained earlier.
2) Next, we use the rate constraint on the conditionally invoked operation as the additional information about frequency of invocation of the condition. That is, the rate constraint is used as a *property of the environment* in continuing the rate constraint analysis to operations that lie above the constrained operation in the graph hierarchy. This way, constraints can be considered as a source of additional information about the system environment, which is considerably more convenient to specify than probabilities of conditions taken. An alternative approach would be to use simulations to collect data, on the likelihood of the condition being true, and use it to derive constraint satisfiability. This approach is out of scope of the this paper.

Recall that the actual execution delay or the latency $\lambda(G)$ refers to the delay of the longest path in G. Unlike length, delay of a path in G may not be bounded in presence of $\mathcal{ND}$ operations. We examine the two cases separately.

B. Case I: G Contains No $\mathcal{ND}$ Operations

The latency of G takes one of the finite values corresponding to interval specified by $\underline{\ell}(G)$. Equations (1)–(6) define the formulae for calculation of $\underline{\ell}$. An upper bound on the operation interval is then given by

$$\max_k (t_k(v_i) - t_{k-1}(v_i)) \le \max_k [\gamma_{k-1}(G) + \lambda_{k-1}(G)] \le \max_k \gamma_k(G) + \ell_M(G). \tag{23}$$

The overhead $\gamma_k(G)$ represents the delay $[t_{k+1}(v_0(G)) - t_k(v_N(G))]$ and can be thought of as an additional delay operation in series with the sink operation, $v_N(G)$. If G is not a root-level flow graph, then there exists a parent flow graph G_+ that calls G by means of a link operation, say v. The upper bound on this interval is derived when the kth and $(k+1)$th invocations of G correspond to separate invocations of the link operation $v \in V(G_+)$. That is

$$\gamma_k(G) = t_{k+1}(v_0(G)) - t_k(v_N(G)) \le t_{j+1}(v) - t_j(v) - x_j \cdot \lambda_k(G) \le \max_j [t_{j+1}(v) - t_j(v)] - \min_j x_j \cdot \min_k [\lambda_k(G)] \tag{24}$$

where x_j is the number of times the flow graph G is invoked for the jth execution of operation v. By definition, G is invoked at least once for each execution of v, i.e., $\min_j x_j = 1$. Therefore, from inequalities (23) and (24)

$$\gamma_k(G) \leq \underbrace{[\ell_M(G_+) + \bar{\gamma}(G_+)] - \ell_m(G)}_{\triangleq \bar{\gamma}(G)}. \tag{25}$$

Note that by definition, $\ell_M(G_+) \geq \ell_M(G) \geq \ell_m(G)$, therefore, $\bar{\gamma}$ is always a positive quantity.

Example 5.2: Consider the graph hierarchy shown in Fig. 2 assuming a purely hardware implementation with no overhead due to the runtime system. We have $\underline{\ell}(G_1) = (1)$ and $\underline{\ell}(G_2) = (3, 5)$. Therefore

$$\bar{\gamma}(G_1) = \ell_M(G_2) + 0 - \ell_m(G_1) = 4.$$

Thus, the overhead in execution of G_1 which is called by a link vertex in G_2 is four cycles.

Lemma 5.1—Minimum Rate Constraint with No $\mathcal{ND}$: A minimum rate constraint on an operation $v_i \in V(G)$, where G contains no $\mathcal{ND}$ operations is satisfiable if

$$\bar{\gamma}(G) + \ell_M(G) \leq \frac{\tau}{r_i} \tag{26}$$

where the overhead term $\bar{\gamma}(G)$ is defined by (25).

Proof: The proof follows from inequalities (20), (23), and (25). □

A bound on the overhead delay $\gamma_k(G)$ can be used to determine bound on the overhead delay of G_+ by examining possible execution paths in G_+. By induction, this process can be carried further to determine a bound on the overhead delay in the invocation of the parent process graph G_o. This overhead delay corresponds to a bound on the delay due to the runtime scheduler overhead. This places restrictions on the choice of the runtime scheduler such that a bound on the scheduling interval can indeed be placed. *Note that a bound on $\gamma_k(G)$ does not necessarily imply a bound on the latency of G.* This is illustrated by Example 5.3 below. An immediate consequence of the above (sufficient) condition for satisfiability of minimum rate constraint is that the question about the constraint satisfiability can be *propagated* as a minimum rate constraint on the link operation in the parent graph model.

Lemma 5.2—Constraint Propagation: A flow graph G satisfies a minimum rate constraint r_i if for its parent graph G_+

$$\bar{\gamma}(G_+) + \ell_M(G_+) \leq \left[\frac{\tau}{r_i} - \Delta\ell(G)\right] \tag{27}$$

where $\Delta\ell(G) \triangleq \ell_M(G) - \ell_m(G)$.

Proof:

$$\bar{\gamma}(G_+) + \ell_M(G_+) \leq \frac{\tau}{r_i} - \Delta\ell(G)$$
$$\Rightarrow \bar{\gamma}(G) + \ell_m(G) \leq \frac{\tau}{r_i} - (\ell_M(G) - \ell_m(G)) \quad [25]$$
$$\Rightarrow \bar{\gamma}(G) + \ell_M(G) \leq \frac{\tau}{r_i}$$

From Lemma 5.1, G satisfies minimum rate constraint r_i. □

In order to obtain a bound on the runtime scheduler overhead, inequality (25) can be unrolled until the parent graph corresponds to the (unconditionally invoked) process model, G_o for which $\gamma(G_o) = \gamma_o$. Thus

$$\bar{\gamma}(G) = \sum_{G_i = G_+}^{G_o} \Delta\ell(G_i) + \bar{\gamma}_o + [\ell_M(G_o) - \ell_m(G)] \tag{28}$$

where $\bar{\gamma}_o = \bar{\gamma}(G_o) = \max_k \gamma_k(G_o)$ is the bound on the delay due to the runtime scheduler. The following example shows the use of constraint propagation in deriving bounds on the delay of the runtime scheduler.

Example 5.3—Minimum Rate Propagation: Consider the hierarchy of graph models used in Example 2.2. reproduced in Fig. 6. Here

$$\begin{aligned} &\underline{\ell}(G_1) = (3, 4) && \Delta\ell(G_1) = 4 - 3 = 1 \\ &\underline{\ell}(G_2) = (9, 10, 14, 15) && \Delta\ell(G_2) = 15 - 9 = 6 \\ &\underline{\ell}(G_3) = (13, 14, 18, 19, 20, \\ &\qquad 21, 23, 24, 25, 26) && \Delta\ell(G_3) = 26 - 13 = 13. \end{aligned}$$

First, we show the intuition behind rate constraint satisfiability, followed by the use of constraint propagation to achieve the same result.

A minimum rate constraint is specified on operation "A" in G_1 that constitutes the loop body of operation two in G_2 with loop index, x, which in turn is a loop body of operation three in G_3. Let $r_A = 1/100$, $r_A^{G_1} = 1/5$, $r_A^{G_2} = 1/25$, and $r_A^{G_3} = 1/50$ cycle^{-1}. Recall, that r_A^G refers to a minimum rate constraint *relative to* G.

Let us first consider $r_A^{G_1} = 1/5$ cycle^{-1}. Since this constraint is relative to G_1, there is no overhead the in invocation of G_1, i.e., $\bar{\gamma}(G_1) = 0$. Since

$$[\bar{\gamma}(G_1)] + \ell_M(G_1) = 4 \leq 1/\frac{1}{5} = 5.$$

The constraint $r_A^{G_1} = 1/5$ is satisfied. Similarly, constraint $r_A^{G_2} = 1/25$ is satisfied since

$$\begin{aligned} \bar{\gamma}(G_1) + \ell_M(G_1) &= [\ell_M(G_2) + \bar{\gamma}(G_2) - \ell_m(G_1)] + \ell_M(G_1) \\ &= [15 + 0 - 3] + 4 = 16 \leq 1/\frac{1}{25} = 25. \end{aligned}$$

Constraint $r_A^{G_3} = 1/50$ is satisfied since

$$\begin{aligned} &\bar{\gamma}(G_1) + \ell_M(G_1) \\ &\quad = [\Delta\ell(G_3) + \Delta\ell(G_2) + \bar{\gamma}(G_3) \\ &\qquad + \ell_m(G_3) - \ell_m(G_1)] + \ell_M(G_1) \\ &\quad = [13 + 6 + 0 + 13 - 3] + 4 = 33 \leq 1/\frac{1}{50} = 50. \end{aligned}$$

Finally, for the minimum rate constraint, $r_A = 1/100$, we should also consider the overhead $\bar{\gamma}_o$ of the runtime scheduler which adds to the bound of 33 cycles on successive intervals of operation "A" *relative to* G_3. Therefore, a r_A is satisfied if the delay due to the runtime scheduler is less than or equal to $100 - 33 = 67$ cycles.

Alternatively, $r_A = 1/100$ can be propagated as a rate constraint of $1/(100 - 1) = 1/99$ on G_2 which is in turn propagated as a rate constraint of $1/(99 - 6) = 1/93$ on

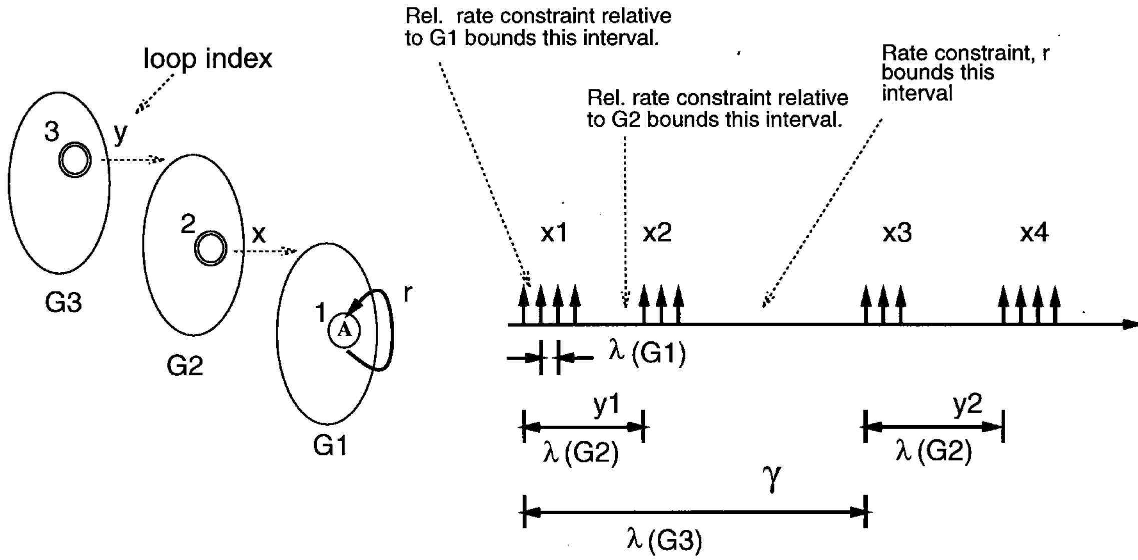

Fig. 6. Upward propagation of minimum execution rate.

G_3. This constraint on G_3 is satisfied for a bound of $93 - \ell_M(G_3) = 93 - 26 = 67$ cycles on the delay due to the runtime scheduler.

Theorem 5.4—Minimum Rate Constraint with No $\mathcal{ND}$: A minimum rate constraint on operation, $v_i \in V(G)$, where G contains no $\mathcal{ND}$ operations is satisfiable if the overhead due to the runtime scheduler is bounded as follows in (29), shown at the bottom of the page.

Proof: The proof is shown at the bottom of the next page.

In summary, a minimum execution rate constraint on a graph model G that contains no $\mathcal{ND}$ operations is translated as an upper bound γ_M on the delay of the runtime system which checked by comparing it against $\bar{\gamma}_o$.

Note, that if the graph G is not a root level graph, then there exists a parent graph G_+ with a link operation that calls G. *However, the unbounded delay due to this $\mathcal{ND}$ operation does not affect satisfiability of the minimum rate constraints on the operations in G.* This is illustrated by the Example 5.3 above where the satisfiability of minimum rate constraints on operation "A" in G_1 is not affected by the $\mathcal{ND}$ operations v_2 and v_3 in G_2 and G_3, respectively. In general, the delay of an $\mathcal{ND}$ operation affects satisfiability of a minimum rate constraint applied on an operation *other than* the operations linked with the $\mathcal{ND}$ operation. This case is considered next.

C. Case II: G Contains $\mathcal{ND}$ Operations

In presence of $\mathcal{ND}$ operations in G, the latency, $\lambda(G)$ can no longer be bounded by the longest path length $\underline{\ell}$ in G. In addition, if G is not a root-level flow graph, its the overhead $\gamma(G)$ may also not be bounded by the maximum path length of its parent graph. For the sake of simplicity, let us first consider the (relative) minimum rate constraint on a graph model with zero overhead, that is, $\gamma_k(G) = 0$ for all $k > 0$. Such a rate constraint then bounds the latency of the graph model and is represented as a backward edge (that is, a maximum delay constraint) from the sink vertex to the source vertex in the constraint graph model of G. Since G is a connected graph, such a constraint invariably leads to a $\mathcal{ND}$-cycle in the constraint graph. According to Theorem 5.2, the maximum delay constraint can be satisfied only by bounding the delay of the $\mathcal{ND}$ operation, that is, by transforming the $\mathcal{ND}$ operation into a non$\mathcal{ND}$ operation. The implications of a bound on the $\mathcal{ND}$ operation delays are as follows.

- Let us first consider synchronization related $\mathcal{ND}$ operations. Since there are multiple ways of implementing a synchronization operation, the effect of the bound is to choose those implementations which are *most likely* to satisfy the minimum rate constraint. Thus, a bound on the delay of the synchronization refers to a bound on the delay offered by the *implementation* of the $\mathcal{ND}$ operation. The implementation delay of a synchronization operation is referred to as the *synchronization overhead* γ_w. This overhead delay is determined by the particular hardware or software implementation of the runtime scheduler. In the absence of a runtime scheduler, in hardware, for example, where the schedule of all operations is statically fixed, the source operation is scheduled for execution immediately after completion of execution of the sink operation, and this overhead

$$\bar{\gamma}_o \le \underbrace{\frac{\tau}{r_i} - \ell_M(G) - \sum_{G_i=G_+}^{G_o} [\ell_M(G_i) - \ell_m(G_i)] - [\ell_M(G_o) - \ell_m(G)]}_{\gamma_M} \quad (29)$$

is zero. For software γ_w, delay is determined by the implementation of the wait operation by the runtime scheduler. For example, a common implementation technique is to force a *context switch* in case an executing program enters a wait state. Here, γ_w would be twice the context-switch delay to account for the round-trip delay. For such an implementation, the minimum rate constraint is interpreted as the rate supportable by an implementation. With this interpretation, the $\mathcal{ND}$ operations are considered non$\mathcal{ND}$ operations with a fixed delay γ_w.

- Next, the data-dependent loop operations use a data-dependent *loop index* that determines the number of times the loop body is invoked for each invocation of the loop operation. The delay offered by the loop operation is its loop index times the latency of the loop body. As mentioned earlier, at the leaf-level of graph hierarchy, the latency of the loop body is given by its path length vector. The elements of a path length vector consist of the lengths of all paths from source to sink, and these are bounded. In case the constrained graph model contains at most one loop operation v, the minimum rate constraint can be seen as a bound on the number of times the loop body G_v corresponding to the loop operation v is invoked. This bound on loop index x is given by (30) that is derived later. This bound $\bar{x}$ is then treated as a property of the loop operation, consequently making it a non$\mathcal{ND}$ operation with a bounded delay for carrying out further constraint analysis. *Verification of these bounds requires additional input from the user*, i.e., the information modeled by the input description in HardwareC is not sufficient to answer the question about constraint satisfiability, and the user is prompted to verify validity of bound $\bar{x}$.

For a relative minimum rate constraint relative to G, the overhead term, $\bar{\gamma}(G)$, in (30) is assigned zero value. In general, however, the satisfiability of a minimum execution rate constraint also includes a bound on the invocation delay, γ of G, as per (28). Clearly, a bound on $\gamma(G)$ implies a bound on the latency of G_+ which is equivalent to a minimum rate constraint on an operation in G_+. However, this minimum rate constraint does *not* bound the loop index of link operation associated with G. The constraint satisfiability is then continued until G_+ corresponds to a process body, G_o.

The presence of multiple $\mathcal{ND}$ operations in G and G_+ present a more complex case since a minimum rate bounds the effective delay which is now a function of multiple loop indexes. In general, this is a difficult problem to answer deterministically since the use of constraints as a property to determine bounds on loop indexes, as mentioned earlier, is also affected by the order in which the $\mathcal{ND}$ operations are evaluated. One straightforward extension of our deterministic analysis procedure for single $\mathcal{ND}$-operation models to models with multiple $\mathcal{ND}$ operations is to allow only one $\mathcal{ND}$ operation at a time and use programmer input to bound other $\mathcal{ND}$ operations. Since the results are dependent upon the order of evaluation of $\mathcal{ND}$ operations, a constraint analysis procedure that determines satisfiability over all possible order is likely to be computationally expensive. Another possibility is to use statistical information about the program behavior. For instance, Puschner and Koza in [24] describe language constructs that allow user to input information about the behavior of the programs that can be used to improve the quality of maximum time estimation.

Theorem 5.5—Minimum Rate Constraint with $\mathcal{ND}$: Consider a flow graph G with an $\mathcal{ND}$ operation v representing a loop in the flow graph. A minimum rate constraint r_i on operation $v_i \in V(G)$ and $v_i \neq v$ is satisfiable if the loop index x_v indicating the number of times G_v is invoked for each execution of v is less than the bound $\bar{x}_v$

$$\bar{x}_v \triangleq \left\lfloor \frac{\tau r_i^{-1} - \bar{\gamma}(G) - \ell_M(G) + \mu(v)}{\ell_M(G_v)} \right\rfloor + 1 \qquad (30)$$

where $\mu(v)$ refers to the *mobility* of operation v and is defined as the difference in the length of the longest path that goes through v and ℓ_M.[2] G_v refers to the graph model called by

[2] The mobility is computed in $O(|E(G)|)$ time as the difference in starting times of as late as possible (ALAP) and as soon as possible (ASAP) schedules of a deterministic delay flow graph constructed by considering all link vertices to be call link vertices with delay as the maximum path length of the called graphs.

$$\begin{aligned}
&\frac{\tau}{r_i} - \ell_M(G) - \sum_{G_i=G_+}^{G_o} [\ell_M(G_i) \\
&\quad - \ell_m(G_i)] - [\ell_M(G_o) - \ell_m(G)] \geq \bar{\gamma}_o \\
&\Rightarrow \frac{\tau}{r_i} \geq \ell_M(G) + \sum_{G_i=G_+}^{G_o} \Delta\ell(G_i) + [\ell_M(G_o) - \ell_m(G)] + \bar{\gamma}_o \\
&\Rightarrow \frac{\tau}{r_i} \geq \ell_M(G) + \left\{ \sum_{G_i=G_+}^{G_o} \Delta\ell(G_i) + [\ell_M(G_o) - \ell_m(G)] + \bar{\gamma}_o \right\} \\
&\Rightarrow \frac{\tau}{r_i} \geq \ell_M(G) + \bar{\gamma}(G) \qquad [28] \\
&\Rightarrow r_i \text{ is satisfied.} \qquad [\text{Lemma 5.1}]
\end{aligned}$$

TABLE III
RUNTIME OVERHEAD IN CYCLES

Implementation	*Processor*	*Overhead, $\bar{\gamma}_o$, cycles*
Subroutine	'86	728
Coroutine	'86	364
Restricted Coroutine	'86	103
Description by cases	'86	85
Restricted Coroutine	DLX	19
Description by cases	DLX	35

the $\mathcal{ND}$ operation v and the overhead bound $\bar{\gamma}(G)$ is defined by (28).

Proof: The maximum interval between successive executions of operation $v_i \in V(G)$ is given by the maximum latency of G and its maximum overhead $\bar{\gamma}(G)$. [See inequality (20) and the following discussion.] The latency of G is defined as the maximum over the lengths of all paths from source to sink vertices. Let p_v represent the longest path from source to sink that goes through operation v

$$\lambda(G) \leq \ell_M(p_v) + (x_v - 1) \cdot \ell_M(G_v).$$

Note that $\ell_M(G)$ is computed by treating all link vertices as call link vertices, and, therefore, it includes the delay due to one execution of each loop body, hence, the second term in equation above represents the additional component to the latency due to the $(x_v - 1)$ invocations of the loop flow graph G_v.

The length of the longest path from source to sink determines the value of $\ell_M(G)$. The vertex v may or may not lie on the longest path from source to sink operations. This slack between $\ell_M(G)$ and the length of the longest path through v is captured by the mobility $\mu(v)$ of operation v. That is, $\ell_M(p_v) = \ell_M(G) - \mu(v)$.

For satisfiability of constraint r_i, we require that

$$\bar{\gamma} + \max_k \lambda_k(G) \leq \frac{\tau}{r_i}$$
$$\Rightarrow \bar{\gamma} + \{\ell_M(G) - \mu(v) + (x_v - 1) \cdot \ell_M(G_v)\} \leq \frac{\tau}{r_i}$$
$$\Rightarrow x_v \leq \left\lfloor \frac{\frac{\tau}{r_i} - \bar{\gamma}(G) - \ell_M(G) + \mu(v)}{\ell_M(G_v)} \right\rfloor + 1.$$

This provides the bound on every loop index in G. □

Example 5.4—Bound on Loop Index Due to Minimum Execution Rate Constraint: Consider a minimum rate constraint of 0.02/cycle on operation "B" in graph model G_2 shown in Example 5.3. Let the maximum delay due to the runtime scheduler be $\bar{\gamma}_M = 0$ (for example, hardware implementation). The bound on the loop index for operation v_2 is calculated as follows:

$$\bar{\gamma}(G_2) = \Delta\ell(G_3) + \bar{\gamma}_M + \ell_m(G_3) - \ell_m(G_2)$$
$$= 13 + 0 + 13 - 9 = 17$$
$$\bar{x}_2 = \left\lfloor \frac{\tau r_B^{-1} - \bar{\gamma}(G_2) - \ell_M(G_2) + \mu(v_2)}{\ell_M(G_1)} \right\rfloor + 1$$
$$= \left\lfloor \frac{50 - 17 - 15 + 0}{4} \right\rfloor + 1 = 5.$$

With this bound on loop index, the $\mathcal{ND}$ operation v_2 has a bound on its delay of 20 cycles.

On the other hand, a *relative* rate constraint, $r_B^{G_2}$ of 0.02/cycle leads to a bound on loop index of

$$\bar{x}_2 = \left\lfloor \frac{50 - 0 - 15 + 0}{4} \right\rfloor + 1 = 9$$

with this bound the delay of v_2 is less than 36 cycles.

VI. CONSTRAINT ANALYSIS IMPLEMENTATION

Operation-level constraint analysis is implemented as a part of the cosynthesis framework, VULCAN [11], to allow the system designer to explore hardware versus software implementations of a given system model. The operation delays corresponding to hardware implementation are obtained using the high-level synthesis tools [12], whereas software implementation is considered in the context of a specific processor-cost model specified by the designer. To evaluate the effect of the runtime system, we have explored the following three ways to implement the software routines: a) subroutine-based, b) coroutine-based, and c) description-by-cases. Briefly, a subroutine implementation refers to translation of program threads into program subroutines that operate under a global task scheduler. In contrast, a coroutine implementation reduces the overhead by placing routines in a cooperative, rather than hierarchical, relationship to each other. The coroutines maintain a local state and willingly relinquish control of the processor at exception conditions which may be caused by unavailability of data (for example, a data dependency on another thread) or an interrupt. In case of such exceptions, the coroutine switch picks up the processes according to a predefined priority list. Upon resumption, a coroutine execution starts execution from the position where its was detached last. A restricted coroutine implementation reduces the overhead further by suitably partitioning the on-chip register storage between program routines such that program counter is the only register that is saved/restored during an interroutine transfer. Finally, in the description-by-cases, we merge different routines and describe all operations in a single routine. This scheme is simpler than the coroutine scheme. Here, we construct a single program which has a unique state assignment for each synchronization operation. A global state register stores the state of execution of a thread. Transitions between states are determined by the runtime scheduling of different ND operations based on the data received. This method is restrictive since it precludes use of nested routines and requires description as a single switch statement, which in cases of particularly large software descriptions, may be too cumbersome. Table III summarizes program overhead for different implementation schemes. Results are reported for two processors, DLX [26] and the Intel 8086. From these implementations, we see that the overhead due to the runtime system in software varies from approximately 20 cycles to over 700 cycles

depending upon the choice of the processor and the runtime system.

The constraint satisfiability tests are put together in the procedure *check_satisfiability*, shown at the bottom of the page. The input to the procedure is a set of graph models with delay and rate constraints along with a choice of the runtime system. Given a flow graph model G with operation delay and execution rate constraints, the constraint analysis proceeds bottom-up. The leaf-level flow graphs do not contain any loop $\mathcal{ND}$ operations. Its output is null if the constraints are satisfiable, else either G is unsatisfiable or it returns bounds on the delay of ND operations that would make constraints satisfiable. These bounds are then verified by the system designer as being applicable or requiring system redesign.

Thus, the satisfaction of the bounds on delay of $\mathcal{ND}$ operations requires additional information from their implementations (such as context switch delay, possible loop index values) against which the questions about satisfiability of minimum rate constraint can be answered. Because of these bounds, there is now a certain *measure* of constraint satisfiability that approaches certainty as the derived bound approaches infinity. More importantly, having bounds derived from timing constraints makes it possible to seek transformations to the system model which trades off these measures of constraint satisfiability against implementation costs. Under certain conditions, these bounds can be extended by modifying the structure of the flow graphs with $\mathcal{ND}$ cycles [11]. The following illustrates an example where the satisfiability tests successfully return with bounds on the $\mathcal{ND}$ operations.

Example 6.1: For Example 5.3, let us assume the following imposed constraints:
$r_A = 1/100, r_A^{G_1} = 1/6, r_A^{G_2} = 1/40, r_B = 1/50, r_B^{G_2} = 1/30, r_C = 1/200$
$u_{CD} = 12, R_B = 0.5, \bar{\gamma}_M = 20.$

Recall

$$\begin{array}{ll} \underline{\ell}(G_1) = (3,4) & \Delta\ell(G_1) = 1 \\ \underline{\ell}(G_2) = (9,10,14,15) & \Delta\ell(G_2) = 6 \\ \underline{\ell}(G_3) = (13,14,18,19,20, & \\ \qquad 21,23,24,25,26) & \Delta\ell(G_3) = 13. \end{array}$$

There are three main steps to the constraint analysis procedure:

1) construction of the constraint graph which is done by adding forward edges for minimum delay and maximum rate constraints and backward edges for maximum delay and (relative) minimum rate constraints;
2) identification of cycles by path enumeration for each of the backward edges in the constraint graph;
3) propagation of minimum rate constraints up the graph hierarchy.

```
check_satisfiability(G) {
      for v ∈ V(G) {
            if v = loop                                   /* recursively go to leaf-level graph */
                  check_satisfiability(G_v);
      }
▷I    construct G_T                                        /* construct the constraint graph model */
      for each backward edge u in G_T {
▷II         if (cycle-set = find-cycles(G_T)){            /* check for min/max */
                  for Γ ∈ cycle-set {                      /* identify cycles caused by backward edges */
                        if (ℓ_M(Γ) > 0)                    /* find positive length cycles */
                              return (Constraint u is unsatisfiable);   /* not feasible */
                        for v ∈ Γ and v ∈ ND {                   /* identify ND cycles */
                              print δ_v = u − ℓ_M(Γ);
                              bound delay of v = min(ℓ_m(v), δ_v);   /* bound on ND delay using constraints */
                              mark v as non-ND;            /* now treat this delay bound as a property */
                        }
                  }
            }
      }
      s = ⌊ℓ_m(G) − τ · max_i R_i^{-1}⌋                    /* check for max rate */
      if s ≥ 0
            return (G is satisfied);
      else {                                               /* need to add null operations – */
            add NOP with δ = |s|;                          /* – to ensure lower bound on delay */
            update ℓ(G_o);                                 /* modified flow graph */
            check_satisfiability(G);
      }
      if G_+ exists                                        /* check for min rate */
▷III        impose constraint [τ/r_i − Δℓ(G)]^{-1} on link operation in G_+;   /* propagate r_i */
}
```

We show these three steps in this example. The procedure first considers $\mathbf{G_1}$:

$\triangleright$I: In the constraint graph of G_1, there are three backward edges with the following weights:

$$\begin{aligned} r_A^{G_1} &= 1/6 \Rightarrow -6 \\ r_A^{G_2} &= 1/40 \Rightarrow -[40 - \gamma(G_1)|_{\gamma(G_2)=0}] \\ &= -[40 - \gamma(G_2) - \ell_M(G_2) + \ell_m(G_1)] \\ &= -[40 - 0 - 15 + 3] \\ &= -28 \\ r_A &= 1/100 \Rightarrow -[100 - \gamma(G_1)] \\ &= -(100 - [\gamma(G_2)] - 15 + 3) \\ &= -(88 - [\Delta\ell(G_3) + \bar{\gamma}_M + \ell_m(G_3) - \ell_m(G_2)]) \\ &= -(88 - [13 + 20 + 13 - 9]) \\ &= -51 \end{aligned}$$

$\triangleright$II: The maximum forward path length is $4 < 6$

$\Rightarrow$ no positive cycles

$\Rightarrow$ The constraints are feasible. Further, G_{T1} contains no $\mathcal{ND}$ cycles.

$\triangleright$III: Propagate minimum rate constraints to

$$G_2 \Rightarrow \begin{array}{ll} r_A^{G_1} & \Rightarrow \text{ not propagated.} \\ r_A^{G_2} & \Rightarrow r_{v_2}^{G_2} = 1/(28-1) = 1/27 \\ r_A & \Rightarrow r_{v_2} = 1/(51-1) = 1/50. \end{array}$$

Note that $r_A^{G_1}$ is not propagated further than G_1.

For $\mathbf{G_2}$:

$\triangleright$II: In the constraint graph of G_2, there are four backward edges with the following weights:

$$\begin{aligned} r_B &= 1/50 \Rightarrow -(50 - \bar{\gamma}(G_2)) = -(50 - 37) = -13 \\ r_{v_2}^{G_2} &= 1/27 \Rightarrow -27 \\ r_{v_2} &= 1/50 \Rightarrow -50 \\ r_B^{G_2} &= 1/30 \Rightarrow -30. \end{aligned}$$

$\triangleright$II: r_B is infeasible since it leads to a positive cycle with weight $= 15 - 13 = 2$. The rest are feasible. Next, the constraint graph contains $\mathcal{ND}$ cycles with a single $\mathcal{ND}$ operation, v_2, for each of the three (feasible) backward edges. Of these, only one, namely $r_B^{G_2}$, bounds the delay due to the $\mathcal{ND}$ operation by the following upper bound on loop index, $\bar{x}(v_2) = \lfloor \frac{30-0-15+0}{4} \rfloor + 1 = 4$. With this bound, the delay of the loop operation, v_2, is bound below 16 cycles.

$\triangleright$III: Propagate minimum rate constraints to

$$G_3 \Rightarrow \begin{array}{ll} r_B & \Rightarrow \text{ Infeasible. Not propagated.} \\ r_{v_2}^{G_2} = 1/27 & \Rightarrow \text{ Not propagated.} \\ r_{v_2} = 1/50 & \Rightarrow r_{v_3} = 1/(50-6) = 1/44 \\ r_B^{G_2} = 1/30 & \Rightarrow \text{ Not propagated.} \end{array}$$

Finally for $\mathbf{G_3}$:

$\triangleright$I: In the constraint graph of G_3, there are three backward edges with following weights:

$$\begin{aligned} u_{CD} &= 12 \Rightarrow -12 \\ r_C &= 1/200 \Rightarrow -(200 - \bar{\gamma}(G_3)) = -180 \\ r_{v_3} &= 1/44 \Rightarrow -44. \end{aligned}$$

$\triangleright$II: There are no positive cycles, so the constraint graph is feasible. Further, two backward edges lead to $\mathcal{ND}$ cycles. Only one of them r_C constrains the delay of the $\mathcal{ND}$ operation v_3. The bound on the loop index, $\bar{x}_3 = \lfloor \frac{180 - \ell_M(G_3) + \mu(v_3)}{\ell_M(G_2)} \rfloor + 1 = 11$. With this bound the delay of v_3 is ≤ 165.

$\triangleright$III: There is no parent graph to propagate the minimum rate constraints.

From Example 5.3, the bound on the delay due to runtime system is $\min\{67, 200 - 13\} = 67$ cycles which is greater than $\bar{\gamma}_M = 20$ cycles. Therefore, with the given bounds on loop indexes of v_2 and v_3 the constraints are satisfied.

This example illustrates the process of constraint analysis for a given hierarchy of graphs that model an embedded system implementation along with the timing constraints. This analysis is interactively performed where the expected bounds on loop indexes are verified by the user to be either acceptable or else loop transformations are attempted to ensure constraint satisfiability [11].

VII. Summary and Future Work

In this paper, we have considered the two basic types of constraints that are useful in characterizing embedded system timing performance. One is operation *delay* constraints which are defined as binary relations that are translated to bounds on the interval between the *start time* of two operations. The other is execution *rate* constraints, typically known as throughput constraints, which are defined as constraints on successive executions of an operation. A notion of constraint satisfiability is developed based on the ability to determine *existence* of a schedule of operations that meets the constraints. Scheduling is considered in two parts: operation scheduling and task/graph scheduling. While the former can be subject to deterministic constraint satisfiability analysis, such analysis for the latter is limited in applicability due to the additional nondeterminism inherent in the runtime system for software. The run-time scheduler models uncertainty in invocation of graph models and, thus, attempts to "merge" this uncertainty with that of delay of (link) $\mathcal{ND}$ operations by choosing an implementation of $\mathcal{ND}$ operation that causes a context-switch. This merge in uncertainty is accomplished by redefining short-term constraint satisfiability over *active* computation times rather than total execution times. Thus, an $\mathcal{ND}$ operation is transformed into a fixed-active-delay operation while the uncertainty associated with its actual delay is delegated to the runtime scheduler.

The proposed algorithm is useful in carrying out an interactive analysis of constraints, where the effect of individual constraints on feasibility of an implementation can be seen by propagating the constraint through the hierarchy of the graph model. The constraint analysis described in this paper has been implemented in a practical cosynthesis system, VULCAN, which takes an input described in HardwareC and allows the designer to interactively explore the effect of individual timing constraints on system implementation in hardware and software. From our experience in applying cosynthesis techniques, we find that the specification of pair-wise operation constraints is sometimes restrictive in real life applications. Our future plans include extension of constraint analysis to

$k$ary constraints, i.e., constraints on $k > 2$ operations. Such constraints are useful in expressing performance constraints on a group of operations that would otherwise require a cumbersome binary constraint descriptions.

ACKNOWLEDGMENT

The authors thank their anonymous reviewers for their comments and constructive suggestions.

REFERENCES

[1] M. Chiodo, P. Giusto, A. Jurecska, A. S. Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, pp. 26–36, Aug. 1994.

[2] D. E. Thomas, J. K. Adams, and H. Schmit, "A model and methodology for hardware-software code-sign," *IEEE Des. Test Comput.*, pp. 6–15, Sept. 1993.

[3] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Des. Test Comput.*, pp. 64–75, Dec. 1993.

[4] R. K. Gupta and G. D. Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Des. Test Comput.*, pp. 29–41, Sept. 1993.

[5] R. Camposano and A. Kunzmann, "Considering timing constraints in synthesis from a behavioral description," in *Proc. Int. Conf. Comput. Des.*, 1986, pp. 6–9.

[6] D. Ku and G. D. Micheli, "Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits," *IEEE Trans Comput.-Aided Des.*, vol. 11, pp. 696–718, June 1992.

[7] S. M. Burns, "Performance analysis and optimization of asynchronous circuits," Ph.D. Dissertation, Calif. Inst. Technol., CS-TR-91-1, Pasadena, 1991.

[8] J. Magott, "Performance evaluation of concurrent systems using petri nets," *Inform. Processing Lett.*, vol. 18, pp. 7–13, 1984.

[9] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using petri nets," *IEEE Trans. Software Eng.*, vol. SE-6, no. 5, pp. 440–449, 1980.

[10] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello, "An algorithm for exact bounds on the time separation of events in concurrent systems," *IEEE Trans. Comput.*, vol. 44, pp. 1306–1317, Nov. 1995.

[11] R. K. Gupta and G. D. Micheli, "A co-synthesis approach to embedded system design automation," *Des. Automat. Embedded Syst.*, vol. 1, nos. 1–2, pp. 69–120, Jan. 1996.

[12] D. Ku and G. D. Micheli, *High-Level Synthesis of ASIC's under Timing and Synchronization Constraints*. Norwell, MA: Kluwer, 1992.

[13] V. Cerf, "Multiprocessors, semaphores and a graph model of computation," Ph.D. Dissertation, Univ. California, Los Angeles, Apr. 1972.

[14] R. K. Gupta, "Co-synthesis of hardware and software for digital embedded systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, Dec. 1993.

[15] R. K. Gupta, C. Coelho, and G. D. Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proc. 29th Des. Automat. Conf.*, June 1992, pp. 225–230.

[16] C. Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Syst.*, vol. 5, no. 1, pp. 31–62, Mar. 1993.

[17] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. Des. Automat. Conf.*, June 1995, pp. 456–461.

[18] D. Bustard, J. Elder, and J. Welsh, *Concurrent Program Structures*. Englewood Cliffs, NJ: Prentice-Hall, 1988, p. 3.

[19] B. Dasarathy, "Timing constraints of real-time systems: Constructs for expressing them, method of validating them," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 80–86, Jan. 1985.

[20] D. Ku and G. D. Micheli, "Relative scheduling under timing constraints," in *Proc. 27th Des. Automat. Conf.*, Orlando, FL, June 1990, pp. 59–64.

[21] I. Watson, "Architecture and performance (fundamentals of dataflow)," in *Distributed Computing*, F. B. Chambers, D. A. Duce, and G. P. Jones, Eds. New York: Academic, 1984, pp. 21–32.

[22] D. Filo, D. C. Ku, and G. De Micheli, "Optimizing the control-unit through the resynchronization of operations," *Integr. VLSI J.*, vol. 13, pp. 231–258, 1992.

[23] Y. Liao and C. Wong, "An algorithm to compact a VLSI symbolic layout with mixed constraints," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, pp. 62–69, Apr. 1983.

[24] P. Puschner and C. Koza, "Calculating the maximum execution times of real-time programs," *J. Real-Time Syst.*, vol. 1, pp. 159–194, Apr. 1989.

[25] G. D. Micheli, D. C. Ku, F. Mailhot, and T. Truong, "The olympus synthesis system for digital design," *IEEE Des. Test Mag.*, pp. 37–53, Oct. 1990.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Los Angeles, CA: Morgan-Kaufman, 1990.

Rajesh K. Gupta received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1984, the M.S. degree in electrical engineering and computer science from University of California, Berkeley, in 1986, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1993.

From 1986 to 1989, he was at Intel Corporation, Santa Clara, CA, where he worked on VLSI design at various levels of abstraction as a member of the design teams for the 80386-SX, 80486, and Pentium microprocessor devices. He was as an Assistant Professor at the University of Illinois, Urbana-Champaign, from 1994 through 1996. Since 1996, he has been an Assistant Professor of Information and Computer Science at the University of California, Irvine. He has worked on a number of successful chip designs, ranging from CMOS, BiCMOS, and ECL to high-speed GaAs devices. He is the author of *Cosynthesis of Hardware and Software for Digital Embedded Systems* (Norwell, MA: Kluwer, 1995). He is coauthor of a patent on PLL-based clock circuit. His research interests are in system-level design and computer-aided design for embedded and portable systems, very large scale integration design, computer systems architecture, and organization.

Dr. Gupta serves on the program committees of the Great Lakes Symposium on VLSI, the CODES Workshop, ICCD, ICCAD, and DAC. He was nominated for the NSF Presidential Faculty Fellow Award by the University of Illinois in 1996. He is a recipient of the National Science Foundation Career Award in 1995, the Philips Graduate Fellowship in 1991 and 1992, the Departmental Achievement Awards by the Microcomputer Division, Intel Corporation in 1987 and 1989, the Components Research Award of the Technology Development Division, Intel Corporation, in 1991, the Chancellor's Award for Excellence in Undergraduate Research in 1987, the Joseph Dias Fellowship, University of California, Berkeley, in 1985, and the Dr. David and Sylvia Gale Fellowship, University of California, Berkeley, in 1984.

Giovanni De Micheli (S'82–M'83–SM'89–F'94) received the nuclear engineering degree from Politecnico di Milano, Milan, Italy, in 1979 and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is Professor of Electrical Engineering and Computer Science at Stanford University, Stanford, CA. Previously, he held positions at the IBM T. J. Watson Research Center, Yorktown Heights, NY, the Department of Electronics of the Politecnico di Milano, and Harris Semiconductor, Melbourne, FL. He is author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994), coauthor of *High-Level Synthesis of ASIC's Under Timing and Synchronization Constraints* (Norwell, MA: Kluwer, 1992), and coeditor of *Hardware/Software Co-Design* (Norwell, MA: Kluwer, 1995) and of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation* (Norwell, MA: Martinus Nijhoff, 1986). His research interests include several aspects of the computer-aided design of integrated circuits and systems, with particular emphasis on automated synthesis, optimization, and validation.

Dr. De Micheli was Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively. He was Co-Director of the NATO Advanced Study Institutes on Hardware/Software Co-Design, Tremezzo, Italy, in 1995 and the Logic Synthesis and Silicon Compilation, L'Aquila, Italy, in 1986. He is the Program Chair (for design tools) of the 1996/1997 Design Automation Conference. He received a Presidential Young Investigator Award in 1988. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS Best Paper Award and two Best Paper Awards at the Design Automation Conference in 1983 and in 1993.