Real-Time Systems, 5, 31–62 (1993)

Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths*

CHANG YUN PARK
Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195†

Abstract. This paper describes a method to predict guaranteed and tight deterministic execution time bounds of a sequential program. The basic prediction technique is a static analysis based on simple timing schema for source-level language constructs, which gives accurate predictions in many cases. Using powerful user-provided information, dynamic path analysis refines looser predictions by eliminating infeasible paths and decomposing the possible execution behaviors in a pathwise manner. Overall prediction cost is scalable with respect to desired precision, controlling the amount of information provided. We introduce a formal path model for dynamic path analysis, where user execution information is represented by a set of program paths. With a well-defined practical high-level interface language, user information can be used in an easy and efficient way. We also introduce a method to verify given user information with known program verification techniques. Initial experiments with a timing tool show that safe and tight predictions are possible for a wide range of programs. The tool can also provide predictions for interesting subsets of program executions.

1. Introduction

Our general goals are to develop techniques for predicting the deterministic timing behavior of high-level language programs. Achieving these goals would allow *a priori* analysis of the timing properties of software. In a recent study [20], [17], we introduced a static analysis technique based on source-level timing schema. It gives tight execution time bounds for many programs, but looser ones for some complex programs. An early experiment showed that a main source of loose predictions is the effects of *infeasible paths* which can be derived from the static program structure but can never be executed in practice. The problem of infeasible paths is inherent in a static analysis, which achieves simplicity by ignoring the details of dynamic behavior. We have developed a method for analyzing dynamic program behavior to eliminate infeasible paths of a program, and have used it in conjunction with the static timing prediction technique to predict tighter time bounds of the program.

A timing prediction method for real-time systems should be able to find guaranteed and sufficiently accurate estimations for the best and worst execution times of a program with reasonable cost. This is incompatible with a pure measurement technique, because only the measurements on the best and worst cases can guarantee safety, but they are very expensive or almost impossible in some complicated systems. Our prediction technique gives guaranteed time bounds and provides flexibility to control their accuracy versus prediction cost. The simple static analysis predicts safe timing estimations which are also reasonably

*This research was supported in part by the Office of Naval Research under grant number N00014-89-J-1040.
†The author's current address is EECS Department, University of Michigan, Ann Arbor MI 48109, cypark@eecs.umich.edu 313-936-0370.

accurate in many cases. Dynamic program analysis refines these estimations by eliminating the effects of infeasible paths while keeping safety. The complexity of program analysis and prediction accuracy are proportional to the amount of provided information about the dynamic behavior of a program. Overall prediction cost is scalable with the degree of precision desired in prediction.

We analyze dynamic program paths based on execution information given by a user. We model user information as well as a program as a set of paths, and intersect them to eliminate infeasible paths and to predict the patterns of possible behavior. Using a well-defined interface language, a user describes his/her knowledge of program execution in an easy and verifiable way. Timing prediction is performed by the simple technique based on timing schema of the earlier study, but it can give much tighter results with the help of dynamic path analysis. To validate our idea, we have built a new timing tool expanded with path analysis and have made experiments with sample programs.

In this work, we assume that a program is correct and written in a high-level procedural sequential language, where recursive procedure call is not allowed. (Timing prediction of concurrent programs is not the subject of this paper; some related discussion will be made in the future work in Section 8.) We also assume that a program has finite execution cases (i.e., finite loop iterations). Our method can be applied directly to a program written in a contemporary language, without modifying it with newly extended language features. (In the experiments, we predict the behavior and time bounds of a program written in a C subset, compiled by the GNU C compiler and run on a 68010-based stand-alone SUN2.)

There have been several other studies about predicting the execution times of a program [22], [13], [19]. One of the common problems here is that predictions are loose for complex programs due to infeasible paths. [19] introduced a few language constructs including *marker*, the maximum execution times of a statement. A user can provide information as a part of a program using those language constructs; however, they are not expressive enough to describe some general execution information. Recently, [15] developed a method of program transformation by partial evaluation. Given a general program, a new specialized and so more predictable program is created by a partial evaluator using knowledge from a user or environment. The method is basically the same as ours in the sense of analyzing a program based on execution information, but their information is at the variable level (e.g., the value of a variable) while ours is at the statement or higher level. Although the partial evaluation technique may deduce information automatically, it requires complex program manipulation. It also seems somewhat inflexible in handling partial information. There are also some studies on semantics-preserving program transformation of concurrent programs [14], [23]. However, they focus on simplifying a program based on static semantics to make schedulability analysis efficient, while we focus on analyzing the dynamic behavior of a program to predict more accurate execution times. Predicting execution times with high-level user information was mentioned in a real-time programming language FLEX [11]. A performance analysis tool predicts the performance of a program by integrating measured time data into a parametric model supplied by a programmer. Its predictions with confidence level may derive realistic performance, but they are stochastic performance that cannot guarantee safety. An extreme case is introduced in [3], where a tool provides the times of program components, and a user writes a timing program, which computes the execution times of program using the provided times.

Program analysis has been studied in many areas, and our work shares some ideas with those studies. Code optimization has used control and data flow analysis for a long time, but it basically handles only static behavior based on syntactic information. Test data generation also performs path analysis to select test paths. Infeasible paths are also a problem here because a selected path may be found infeasible while generating data. There are only a few testing studies addressing this problem; one of them used informal *allegations* which are similar to primitive execution information [24]. Symbolic execution has been used in many program analysis studies (e.g., [25]). It is a general way to figure out dynamic behavior of a program, but its complexity makes it almost intractable. [6] introduced a program simplification method, which is somewhat similar to [15]. A program is decomposed into subprograms in a pathwise manner, based on state constraints which are inserted into a program by a user. We decompose the behavior only without modifying a program. Path-oriented static program analysis was mentioned in [16]. In the study, both a program and a constraint are represented by an extended regular expression, and it is proved using dataflow analysis techniques if the program satisfies the constraint. However, the analysis counts only static behavior, and the goal is verification rather than analysis as in our study. *Constrained expressions*, a similar technique at design level, was presented in [1].

Our study shows several interesting points. First, the dynamic behavior of a program can be analyzed with the help of user information. The formal path model with a well-defined interface language provides a systematic way to exploit a user's high-level knowledge. Second, predictions can be very tight even for complex procedures. Adding dynamic path analysis makes it possible for a simple technique based on timing schema to predict tight bounds regardless of the complexity of a program. We can also produce many potentially useful results such as execution patterns and performances of some specific cases. Third, a prediction can be refined by a separate and scalable compensation step. A prediction method can be kept simple ignoring complications beyond statement-level. If more precise predictions are wanted, dynamic path analysis compensates predictions by reconsidering the ignored complications. Since the two processes are independent, path analysis does not degrade the simplicity of prediction. We believe that this approach is more efficient than an approach that makes the prediction method more complex.

Section 2 summarizes our earlier prediction approach and experiments. Section 3 explains user execution information and introduces a formal path model as the theoretical basis of our path analysis. In the next three sections, a practical approach is explained discussing the following problems respectively: how to represent user execution information, how to verify the correctness of information given, and how to process information with a program. Section 7 shows experimental results with a timing tool that implements our timing prediction with dynamic path analysis. We close this paper mentioning future work in the final section.

2. Summary of Timing Schema Approach

Our prediction approach is based on the notion of *timing schema* for source program constructs, which are essentially formulae for computing the lower and upper bounds for their execution times [20]. Timing schema decompose a statement into the component blocks,

and later compute the times of the statement by composing the predictions of those blocks according to the system-independent language semantics. The times of a component block which are dependent on implementation are predicted through object code prediction and machine analysis. Applying these decomposition, prediction and composition recursively, we can compute the time bounds of a statement, compound statement and procedure. The time prediction T of a statement or a block S is represented by a pair of lower and upper bound (i.e., $T(S) = [t_{low}, t_{up}]$), where each bound t_{low} and t_{up} estimates the best and worst execution time, respectively. For example, the execution times of a simple C statement `S1: while (a) a--;` can be predicted as shown in Figure 1. It shows how a statement is decomposed and predicted according to its timing schema on a target system, a GNU compiler and a MC68010 processor in this example.

Statement:

```
S1:  while (a) a--;
```

Timing Schema:

For loop statement S : **while** (exp) stmt ;

$$T(S) = (N+1) \times T(\text{exp}) + N \times T(\text{stmt}) + T(\textbf{while},N)$$

where N is a pair of loop bounds (i.e., $N = [n_{min}, n_{max}]$), provided by a user.

GNU C compiler's code generation rule and code for S1:

```
          start_loop                  L1:
          exp                            tst1  a6@(-4)      ; I1
S  ===>   exit_if_false   S1  ===>       jeq   L2           ; I2
          stmt                           subql #1,a6@(-4)   ; I3
          end_loop                       jra   L1           ; I4
                                      L2:
```

Code Prediction:

T(exp) : T(a) = T(I1)
T(stmt) : T(a--) = T(I3)
T(**while**,N) : $N \times T$(I2,fail) + T(I2,succ) + $N \times T$(I4)

where "I2,fail" is conditional jump instruction I2 whose branch is not taken.

Machine (MC68010) Analysis:

T(I1) = [16 , 16] (clock cycles)
T(I2,fail) = [6, 6]
T(I2,succ) = [10, 10]
T(I3) = [24, 24]
T(I4) = [10, 10]

Time Computation:

Suppose a user gives [0, 1] as loop bounds knowing that a is 0 or 1. Then,

$$\begin{aligned} T(S1) &= (N+1) \times T(\text{a}) + N \times T(\text{a--}) + T(\textbf{while},N) \\ &= (N+1) \times T(I1) + N \times T(I3) + N \times T(I2, fail) + T(I2, succ) + N \times T(I4) \\ &= ([0,1]+1) \times [16,16] + [0,1] \times [24,24] + [0,1] \times [6,6] + [10,10] + [0,1] \times [10,10] \\ &= [26, 82](\textit{clock cycles}) \end{aligned}$$

Figure 1. Timing schema and predicting the execution times of a **while** statement.

We believe that software timing properties should be reasoned about at the source language level, where a program is written, analyzed, debugged and maintained. Even for timing prediction, the lower level (e.g., assembly-level) analysis results in more loss than gain, because high-level information (e.g., algorithm or statement relations) is usually more valuable than low-level one (e.g., code optimization). We also expect that many code optimizations can be handled by adjusting the granularity of decomposition and parameterizing timing schema.

We built a timing tool predicting the execution times of a program written in a subset of C, compiled by GNU C compiler and executed on a MC68010-based standalone SUN2 [17]. The tool could be partitioned cleanly into an abstract system-independent portion and a lower level system-dependent part. The major issues were to determine the granularity of an atomic block covering the effect of compiler's default optimization, and to handle machine nondeterminism such as instructions with variable execution times and unavoidable system interference (clock interrupt and memory refresh). We compared predictions by the tool with measured times of the best and worst execution cases for sample programs. The experiments showed that the simple timing schema approach can provide safe and useful predictions. However, predictions are looser for some complex programs having infeasible paths. Since timing schema cannot handle inter-relations among statements, predictions include the times of infeasible paths. For tighter prediction, the method needed to be extended to take into account of the relations of statements and to eliminate the effect of infeasible paths.

3. Execution Information and Path Model

3.1. Execution Information

While organized with given language constructs, a program may encode some unintended behavior in the structure along with intended behavior. The unintended behavior, however, is prevented at runtime by constraints encoded with program logic and implicit data value assumption. Thus, the program structure only is not sufficient to figure out the real execution behavior of a program. *Execution information* is information about the execution behavior of a program, such as a programmer's intentions and constraints encoded in the program. It can be used to remove some infeasible paths, which are formed in programming but not intended in execution.

We believe that execution information can be supplied by a user, usually a programmer. Since a programmer knows the details about the applied algorithm, program logics and specifications when he/she writes a program, it is not difficult to provide some execution information for the program. In some sense, execution information is another description of a program, explaining how it works in real execution. Some mechanical methods such as symbolic execution may also extract execution information from a program. However, they only work in some restricted cases and are very complex; some user information such as loop bounds is still necessary. User information, as long as it is correct, is usually more refined and flexible to use than one detected in a mechanical way. (Since a user may give wrong information, its correctness must be verified. We will explain it in Section 5.)

Any hints on program execution can be execution information. Examples include an iteration number for a loop statement and relations among statements (e.g., two statements S1 and S2 must be executed together). We observe the following characteristics of user execution information:

- *Incompleteness.* It is unrealistic to expect a user has complete knowledge of every program behavior. Knowledge may also be limited to local behavior in some program segments. A user usually gives multiple statements of *partial* information.
- *High-level.* The basic element of user information is mostly a statement or higher level one (e.g., sequence of statements). Some lower-level information, such as the value of a variable, can be replaced with statement-level information through a manual or an automatic deduction process.
- *Conditional and Scope-dependent.* User information may have conditional clauses (e.g., if S1 is executed, S2 is not executed), and information on the same statement may vary depending on the applied scope (e.g., S1 is executed a maximum of 10 times in loop L1, but never executed more than 100 times in outer loop L2).
- *Interprocedural.* As a program is modularized, it is composed of many small procedures. Thus, some execution information crosses procedure boundaries, giving the relations among procedures.

3.2. Path Model

We define a *path* as a sequence of program statements where a statement is represented by a label. A path is called a *program path* if it is defined in a program, following the control flow of the program from the start to the end. A program path is *feasible* if it is executable on some data set. An *infeasible* path means a program path which is not feasible; that is, it is possible statically, but impossible dynamically. We also use a *subpath* to mean the subsequence of a path.

In our path model, each statement of user execution information is represented by a set of paths in a constructive way. For an information statement, we construct a set of all paths satisfying its intended constraint and thus being feasible with respect to the information only. The set should include all feasible paths of a program, but it excludes infeasible paths which violate the constraint. For example, if a user has information, *statement a is always executed*, it is represented by the set of all paths passing a. The set includes no path not passing through a. No information becomes a set of all paths (i.e., all paths are feasible with no information).

There are two main reasons why we chose the path model. First, a set of paths is a general and uniform way to describe various user execution information. Second, both a program and information are represented in the same language. By representing information with paths, the problem of information processing becomes that of path processing, and we can use well-known operations and results in path processing.

The path model is illustrated in Figure 2. Let A_P be the set of all statically possible program paths, a path representation of program P. A_P includes a set of all feasible paths X_P but also has some infeasible program paths, which our goal is to eliminate. Let I_P be user

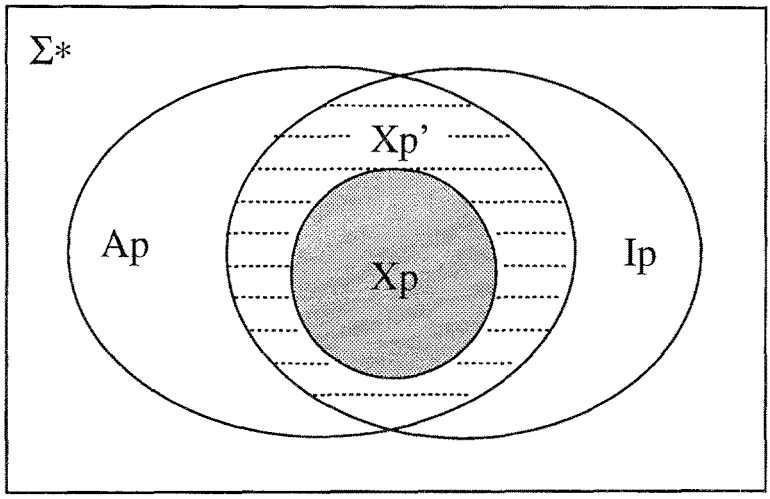


Figure 2. Path model.

execution information of P, also as a set of paths. Since user information is usually partial, I_P is also usually a partial approximation to X_P. It may include some infeasible paths that are not constrained by the information. I_P can also include even nonprogram paths (i.e., the portion of I_P not in A_P), because the information may mention the local behavior of some program segments, not that of the whole program. However, as constructed towards X_P, I_P must include all feasible paths X_P.

The operation of filtering infeasible paths can be achieved by simply intersecting A_P and I_P, because both of them are supersets of X_P, and I_P excludes some infeasible paths. The intersection result, X_P' is a safe estimation of X_P; we call X_P' the set of possibly executable paths. It still may have some infeasible paths not excluded by I_P, but it covers all feasible paths. The portion of I_P not in A_P is also eliminated by A_P. The difference between A_P and $X_P'(A_P - X_P')$ becomes the eliminated infeasible paths. In summary, the relations among the path sets are

$$X_P' = A_P \cap I_P \tag{1}$$

$$X_P \subseteq X_P' \subseteq A_P \tag{2}$$

Relation 2 is also true in timing. For a set of paths Π, let $T(\Pi)$ be a time interval for the execution times of the paths in Π. (We use the same timing notation $T(\Pi)$ for a set of paths as for a program $T(P)$, because a program is represented as a set of program paths in path analysis.) Then,

$$T(X_P) \overset{B}{\subseteq} T(X_P') \overset{B}{\subseteq} T(A_P)$$

where interval inclusion operator $\overset{B}{\subseteq}$ is defined as follows: $[a, b] \overset{B}{\subseteq} [c, d] \equiv (c \leq a) \wedge (b \leq d)$. Thus, $T(X_P')$ is also a safe estimation to the execution time of P, and it is tighter than $T(A_P)$ based on the program structure only.

symbols ::

alphabets(Σ) : a set of statement labels
operators : + , . , * , $\cap$, $\neg$
parenthesis : (,)
empty set : $\emptyset$
wild cards :
* means any string of any statements (Σ^*)
_ means any string of statements not containing adjacent statements
(i.e., $x\_y = x(\Sigma - \{x, y\})^*y$)

path ::
a regular expression of symbols

Figure 3. Path language based on regular expression.

Our language for path description (say *path language*) is *regular expressions* extended with the operators for intersection and negation, as summarized in Figure 3. These two operators do not increase descriptive power, but they allow compact representation of some information requiring an exponential length expression in regular expressions without this extension. We also introduce two wild cards, "*" (different from the Kleene star *) and "_". Figure 4 shows the path descriptions of a program and some user information in our path language. Note that A_P requires neither intersection, negation nor any wild cards. The details of statement naming will be described later (Section 4.2).

The choice of regular expressions was obvious because it is a formal language widely used in many areas including program representation. Its properties are well-known, and most operation and decision algorithms have also been developed. The descriptive power is at least sufficient to describe all feasible paths (X_P), because we assume a program has only finite feasible paths. Other good points are that its wild cards are good at abstraction for partial information, and it (its recursive definition) uniformly treats a statement, a path and a set of paths, which is helpful for handling interprocedural information.

As discussed in Section 3.1, a user usually gives multiple statements of local and partial information. As combined conjunctively, those partial information statements compose more global and specific information for the program. In the path model, each information statement becomes a set of paths (I_P^i), and combining partial information can be done by intersecting the sets of information paths. Equation 1 can be rewritten as follows

$$\begin{aligned} X_P' &= A_P \cap I_P \\ &= A_P \cap (I_P^1 \cap I_P^2 \ldots I_P^n) \\ &= (\ldots((A_P \cap I_P^1) \cap I_P^2)\ldots) \end{aligned}$$

A set of information (I_P^i) is usually described by concatenations of some specific subpaths and nonspecific subpaths. Specific subpaths composed of statement labels represent the meaning of the information. Nonspecific ones make the specific ones into a set of paths

Program

```
check_data()
{
    int  i, morecheck, wrongone;
    morecheck = 1;  i = 0; wrongone = -1;
L:  while (morecheck)
 LB:{
      if (data[i] < 0)
    A:  {  wrongone = i;  morecheck = 0;  }
      else
    B:  if (++i >= datasize)  morecheck = 0;
    }
    if (wrongone >= 0)
   C:  {  handle_exception(wrongone);  return 0;  }
    else
   C': return 1;
}
```

Program Path (A_P)

$$L \cdot (LB \cdot (A + B))^* \cdot (C + C')$$

User Information and I_P

1. The loop is executed 10 times since the size of data is known externally to be 10.

$$\Rightarrow \_(LB\_)^{10}$$

2. If A is executed, the loop is finished and C is executed next.

$$\Rightarrow \neg(*A*) + \_A \cdot C*$$

Figure 4. Path descriptions of simple program and path information.

that includes all feasible paths. Nonspecific subpaths, usually wild cards, may be very abstract, because after intersecting A_P, X_P' has only program paths of P regardless of what I_P^i has.

3.3. Pathwise Decomposition

Path processing in the path model is basically to compute $A_P \cap I_P$ and determine X_P', all of which are described by regular expressions. Since X_P' should tell the behavior of a program and will be used later in timing prediction, it must be a set of program paths without any wild cards. We also remove $\cap$ and $\neg$, because they cause difficulties in timing prediction.

Our principle of path processing is *pathwise decomposition*. As discussed earlier in Section 3.1, a program is the result of structured composition of intended behavior. Better reasoning of a program is possible by decomposing the behavior. In the path model, this can be achieved by pathwise decomposition meaning that all of the possibly executable

paths are decomposed into several subsets where each subset represents a pattern of program behavior. During path processing, we express intersection results as a sum of subsets. Then, after intersecting all information paths,

$$X'_P = \Pi^1_P + \Pi^2_P + \ldots + \Pi^n_P$$

and each subset Π^i_P represents a group of possible executions having common behavior patterns.

Pathwise decomposition is also compatible with timing prediction. First, compute the time bounds of each subset ($T(\Pi^i_P)$) using our basic static timing prediction technique. We may have tighter bounds because each subset describes more specific paths (and thus has fewer infeasible paths) than A_P. Then $T(X'_P)$ is computed as

$$\begin{aligned} T(X'_P) &= T(\Pi^1_P) \cup T(\Pi^2_P) \cup \ldots \cup T(\Pi^n_P) \\ &= [min\{low(T(\Pi^i_P))\},\ max\{up(T(\Pi^i_P))\}] \end{aligned}$$

where *min* and *max* are the minimum and maximum function, and *low*(T) and *up*(T) denote the lower and upper bound of T, respectively, as defined in Section 2.2.

There exists a known algorithm for resolving intersection and negation in regular expressions. However, it was proved that the complexity of the problem requires exponential time in the general case [12]. The length of a regular expression may also increase exponentially after resolving intersection. These complexity results imply that we need to find some practical solutions appropriate to our problem domain.

4. Information Description Language

4.1. Two Level Language Scheme

Our regular expression based path language is compact, and makes it possible to represent a program and user information in the same way. Conceptually, processing is simple enough to be done by one operation, intersection. However, intersection requires exponential complexity in reality. Another problem is that our path language may be too formal for a user. Describing one's knowledge with the operators and wild cards of regular expressions seems to be difficult and error-prone. Finally, we have to develop a method to verify given path information.

We believed that the above problems of the path language can be addressed by compromising the descriptive power for ease of processing and by introducing a high-level user interface language. A user gives execution information in the interface language which is somewhat restrictive but easy to use and verify. The given information is then mechanically transformed into the base language, i.e., the path language. The transformed path information is processed as shown in the path model, but it can be done efficiently because the information now has some restricted forms only allowed in the interface language.

The interface language should be a *practical high-level subclass* of the path language. It must restrict a user to prevent transformed information from being too complex to process. However, it should provide sufficient ways to describe necessary and valuable information. It should also be high-level for ease of use. Finally, it must be formal not only to be able to transformed to be the path language but also to be verifiable from a program text or specifications.

In developing the interface language, we made the following decisions based on the characteristics of user execution information discussed in Section 4.2.2. First, the language is statement-based. A statement is the basic object in the language; the notion of a path may appear in a restricted way, only through path grouping. Second, it is logic-based; information is expressed as logical relations among statements. The above two decisions come from the observation that a sequence may cause difficulty in intersection, and it can be replaced with logical descriptions in most cases. Logic-based information may also be compatible with existing verification techniques based on program logic. Third, information on iteration numbers for a loop and on the execution counts of a significant statement (i.e., how many times it is executed) should be describable. Although they result in complicated intersections, they are necessary in determining the realistic behavior of a program and eventually predicting the times. Finally, to support interprocedural information, we provide a method to summarize the behavior of a procedure and to export it to other procedures.

4.2. Information Description Language (IDL)

Based on the above decisions, we define the information description language (IDL) as an interface language directly accessed by a user. Figure 5 shows some IDL information statements for a procedure. It also shows how interprocedure information between two procedures is expressed.

The basic element is a *statement name*, which stands for a statement, compound statement or procedure path in a program. A name for a statement and compound statement is identified as a label in a program, and a name for a procedure path is defined in the path group information for the procedure (which will be explained later). We also allow *default naming* where a user can address a statement by a *hierarchical* name derived unambiguously from other labeled statements (e.g., *A.then* denotes the statement executed when the condition of *A* is true). A labeled name always overrides the default name. We believe that properly used default naming is easy to understand and reduces troublesome name composition.

Many statements are neither labeled in a program nor mentioned in IDL informations directly or by default. They are totally invisible in program analysis, not because they are less important but because they are executed simply as specified by the program structure or a user has no information on them. Their timing behavior must be kept intact during program analysis and be counted in timing computation of a program.

Table 1 shows the syntax of IDL. IDL information consists of two parts, execution information and path group information. The basic unit of IDL execution information is INFO-CLAUSE, which describes relations among statements and the execution counts of a statement. Path group information part is composed of lines of GROUP-SENTENCE, each of which describes one instance of path grouping.

• **Procedure**

```
check_data()
{
    int  i, morecheck, wrongone;
    morecheck = 1;  i = 0; wrongone = -1;
L:  while (morecheck)
    {
      if (data[i] < 0)
    A:  {  wrongone = i;  morecheck = 0;  }
      else
    B:  if (++i >= datasize)  morecheck = 0;
    }
    if (wrongone >= 0)
   C:  {  handle_exception(wrongone);  return 0;  }
    else   return 1;
}
```

• **Information**

1. **loop** L [1,10] **times**;
   (Loop L is iterated [1,10] times; the size of data is known as 10.)
2. **samepath**(A,C);
   (Statements A and C must be executed together.)
3. (**not** A) **imply loop** L 10 **times** ;
   (If A is not executed, then L is iterated 10 times.)
4. **execute** A[0,1] **times inside** L;
   (The exception case A is executed at most once inside L.)
5. **pathgroup** EXCEPTCASE **passing** C ;
   (All paths passing through C are grouped into EXCEPTCASE.)

• **Another Procedure**

```
task1()
{
   ......
A: dstatus = check_data();   /*  invoke ''check_data''  */
   if (!dstatus)
  B:  clear_data();
   ......
}
```

• **Interprocedural Information**

1. **samepath**(A.check_data.EXCEPTCASE , B) ;
   (Statement C in "check_data" must be on the same path with statement B in "task1".)

Figure 5. Example of IDL information.

Table I. The syntax of IDL.

```
INFORMATION      ::= INFO_LINES | GROUP_LINES
                   | INFO_LINES  GROUP_LINES
INFO_LINES       ::= INFO_LINE
                   | INFO_LINES INFO_LINE
INFO_LINE        ::= INFO_SENTENCE
                   | INFO_LINE or INFO_SENTENCE
INFO_SENTENCE    ::= INFO_CLAUSE [ SCOPE ] ';'

INFO_CLAUSE      ::= ALWAYS | SPATH | NPATH | XPATH
                   | LOOP | TIMES | IMPLY
                   | INFO_CLAUSE or INFO_CLAUSE
                   | INFO_CLAUSE and INFO_CLAUSE

ALWAYS           ::= always '(' STMT [, STMT] ')'
SPATH            ::= samepath '(' STMT , STMT [, STMT] ')'
NPATH            ::= nopath '(' STMT , STMT [, STMT] ')'
XPATH            ::= exclusive '(' STMT , STMT [, STMT] ')'
LOOP             ::= loop STMT CONSTANT times
TIMES            ::= execute STMT CONSTANT times
IMPLY            ::= COND imply INFO_CLAUSE

SCOPE            ::= inside STMT
CONSTANT         ::= INTEGER
                   | '[' INTEGER ',' INTEGER ']'
COND             ::= STMT  | '(' COND ')'
                   | not COND
                   | COND or COND
                   | COND and COND

GROUP_LINES      ::= GROUP_SENTENCE
                   GROUP_LINES  GROUP_SENTENCE
GROUP_SENTENCE   ::= pathgroup STMT passing COND ';'
```

A statement name is also used to specify the *scope*, the boundary where information is effective. As we discussed in Section 3.1, a user information statement may be true in one scope but may be false in other scopes. To avoid confusion, a user can specify the scope of his/her information as the name of the bounding statement. If the scope is not specified explicitly, the *default scope* becomes the nearest common enclosing compound statement of the statements appearing in given information; the only exception is **always** clause where the default scope becomes a procedure. We introduce two default labels S.$s and S.$t for the start and end of non-simple[1] statement *S*. They may not be used in an IDL description, but appear in translated path information as delimiters specifying a scope. These delimiters not only make path information precise without confusion but also make path processing easier by localizing intersection.

Instead of giving a formal semantics for IDL, we here explain its meaning informally. Four of seven INFO_CLAUSEs are constraints on the relations among statements. **always**(*A*) means that statement *A* is always executed in all paths. If two statements are always executed together, a user can say **samepath**(*A*,*B*), meaning *A* and *B* must be on

the same path. Similarly, "**nopath**(A,B)" is used when there must be no path passing both A and B, and "**exclusive**(A,B)" when A is executed if and only if B is not executed.[2] For the execution counts of a statement, two clauses are provided. **loop** L N **times** means that loop statement L iterates N times, and **execute** A M **times** says that A is executed M times inside given scope. Finally, C **imply** I describes conditional information saying if condition C is true then information I is true. A statement appearing in condition (COND) means the execution of the statement. For example, (A) **imply always**(B) means that if A is executed, then B is executed. The meanings of higher-level constructs are obvious. As an example, "**nopath**(A) **inside** $L1$; **or execute** A 1 **times inside** $L2$;" means that statement A is not executed inside $L1$ or it is executed only once inside $L2$.

Path group information does not eliminate infeasible paths; it is applied to organize the results of path processing and to express interprocedural information. After all execution information is processed in a pathwise manner. X'_P is decomposed into several subsets. However, this decomposition does not always provide a summary that a user wants to know; it may be too coarse or too refined to extract the patterns of behavior. Thus, we provide a user with ability to define cases that he/she wants to know. A pathgroup information **pathgroup** PATHNAME **passing** COND has the meaning that all paths in X'_P satisfying COND are grouped and named as PATHNAME. In IDL a condition clause is restricted to have only statements conneced by logical operators.

One purpose of path grouping is to analyze the execution patterns of a procedure. In addition to the execution time bounds of all cases, a user may be interested in case by case analysis. (For example, for the procedure ''check_data'' in Figure 5, a user may want to know the execution times for the case when an exception happens.) Through path grouping, one can define some execution cases of a procedure and have predictions on them, the behavior as paths and the timing as bounds. With some qualification, each path group becomes a more specific representative of the procedure. Thus, we treat it like a separate procedure whose name is extended with the path group name as a qualifier (i.e., check_data.EXCEPTCASE).

The other purpose is to simplify handling interprocedural information. The easiest way to describe interprocedural information is to access the statement names of other procedures directly as the statements in the same procedure. However, it implies that the procedures should be processed together. A procedure should be analyzed whenever it is called, and the size of path representation grows. Path grouping provides an indirect (but structured) way to access statements across procedures. If all paths passing through a statement are grouped by path group information, interprocedural information to a statement may be replaced with information to the path group name. Since the behavior of the path group has already been analyzed, we simply use the exported results without processing the procedure again. Certainly, interprocedural information with path group names is less expressive than information described directly with statement names, because a condition clause in path group information only has statements. We expect, however, that many useful interprocedural information statements can be expressed with path group names.

4.3. IDL Translation

User information written in IDL is translated into regular expressions which represent the set of information paths (I_P) in the path model. Translation is done following the structure

of IDL; IDL information is translated line by line, and inside a line it is done in bottom-up fashion from statements to clauses and to sentences.

The summary of translation rules is given in Table 2. The basic idea in translation is to construct a set of paths satisfying the constraints meant by an IDL statement; all feasible paths (X_P) must be included into the set because they certainly satisfy the constraints. Regular expressions for an information clause can be built easily according to their logical semantics. For an information sentence, its scope, determined by default if not specified explicitly, should be translated. Since a scope is basically a condition that the clause is true only if the scope statement is executed, we have to add paths where the scope is not executed. For the case that the scope is executed, its paths are constructed as a Kleene closure of the translated clause with the distinguished delimiters for the start and end of the scope. (We call this closure a *scope closure*.)

It is worthwhile to note that translated regular expressions have only a few patterns. Of course, this is the goal to introduce a restrictive interface language. Since information paths are not so general but somewhat structured now, we can find an efficient algorithm

Table 2. IDL translation rules.

1. **Statement names in condition clause**

   $A \Rightarrow (*A*)$

2. **Logical operators**

   and $\Rightarrow \cap$
   or $\Rightarrow +$
   not $\Rightarrow \neg$

3. **Information clause**

   (a) **always**(A)
   $\Rightarrow (*A*)$

   (b) **samepath**(A,B)
   $\Rightarrow (*A*) \cap (*B*) + \neg(*A*) \cap \neg(*B*)$

   (c) **nopath**(A,B)
   $\Rightarrow \neg((*A*) \cap (*B*))$

   (d) **exclusive**(A,B)
   $\Rightarrow (*A*) \cap \neg(*B*) + \neg(*A*) \cap (*B*)$

   (e) **loop** A K **times**
   $\Rightarrow \neg(*A*) + \_\ A.\$s\ A.\$c\ (\_\ A.\$c)^K\ A.\$t\ \_$
   where $A.\$c$ denotes the default name for the loop condition of A.

   (f) **execute** A K **times**
   $\Rightarrow (\_\ A\ \_)^K$

   (g) C **imply** I_C
   $\Rightarrow \neg\sigma_C + \sigma_C \cap \sigma_{I_C}$
   where σ_C *and* σ_{I_C} represents the translated regular expressions of C and I_C, respectively.

4. **Information Sentence**

   I_C **inside** S ;
   $\Rightarrow \neg(*S*) + (\_\ S.\$s\ \sigma_{I_C}\ S.\$t\ \_)*$
   where $S.\$s$ and $S.\$t$ are special default statement labels for the start and end of S. The scope S is determined by default if it is not given explicitly.

intersecting them with program paths (A_P). Here, we emphasize again that IDL is one of many possible subclasses of path language. We chose IDL by trading off between expressiveness and processibility based on the characteristics of user information. The appropriateness of IDL can be validated through a lot of experimental work. (We will address it in Section 7.)

5. Verification of User Information

Since we eliminate paths in prediction based on user information, the correctness of given user information is critical to the correctness of predicted results. If a user gives wrong information, the tool may eliminate some executable paths, and predictions neglecting those paths may be unsafe. What does it mean that information is correct? We say that user information is correct if every execution of a program follows its semantic or constraints. (In the formal path model, I_P is correct if $X_P \subseteq I_P$.) To prove correctness, all execution cases should be reasoned about from a program text and all preconditions on which the program is running. Preconditions (e.g., the domain of data values) may be not specified explicitly in a program, but imposed implicitly by specifications such as assumptions on the environment.

In this work, we do not introduce a technique or a tool to check the correctness of given user information automatically. It is already known that proving a properties of a program is undecidable in general. There have been some program verification tools (e.g., [16]), but our user information is more sophisticated than their capability because it specifies high-level dynamic properties. Our goal is to provide a user with a formal way to reason about the correctness of his/her information. Actual verifiction should be made by a user according to the procedure. The message here is that our user information is sufficiently formal to prove its correctness, and verification can be done by well-known techniques.

The basic strategy is to use an assertional program logic; Figure 6 illustrates the procedure. First, user information for a program is transformed to an assertion on program variables. Then by proving that the assertion is true in the extended program with a proper program logic, such as Hoare logic [5], or Dijkstra's weakest-precondition [4], we can verify the correctness of the information. Some assertions cannot be verified by a program text only, because they are related to some assumptions on the system and/or environment. Knowledge of those assumptions provided from specifications is transformed to preconditions at the beginning of the program. We chose program logic because it is a well-known powerful technique for program verification. Since our IDL is logic-based, the transformation is also easy.

Figure 7 shows how a problem of verifying IDL information is reduced to a problem of program verification. Since information I is about statement A and B with scope L, program P is extended by adding 3 statements (enclosed by $< >$) after A and B and before L. The assertion to be verified is put at the end of L.

Equivalence relation between the information (I) and the assertion (A_I) is obvious. Since $\#A$ and $\#B$ are zero before L and they are incremented only when A and B are executed, the value of $\#A$ and $\#B$ after L represents the number of times A and B are executed inside L, respectively. Thus, the assertion A_I is true if and only if both of A and B or neither

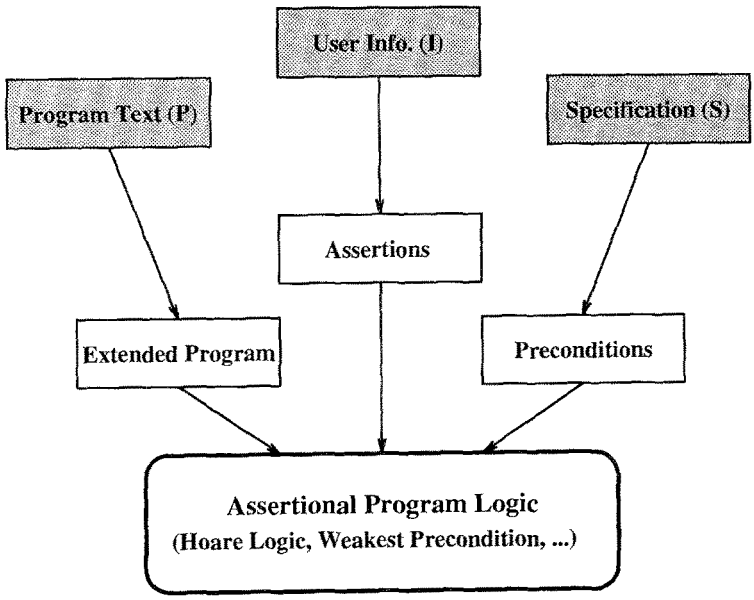


Figure 6. Verification procedure.

- **Program**

```
P ::
begin
   ...
L  begin
     ...
     A: S_A
        ...
        B: S_B
     ...
   end
end
```

- **User Information**

```
I :: samepath(A,B)inside L ;
```

- **Assertion and Extended Program**

```
P'
begin
   ...
   < #A = #B = 0 ; >
L: begin
     ...
     A: S_A     < #A = #A + 1 ; >
        ...
        B: S_B     < #B = #B + 1 ; >
     ...
   end
        { A_I = (#A > 0 ∧ #B > 0) ∨ (#A = 0 ∧ #B = 0) }
end
```

Figure 7. An example of problem reduction.

of them are executed throughout L; that is, A and B are on the same path. Assertions for other types of information can be easily built using predicates on the value of auxiliary variables, in a similar way to the IDL translation. For example, A **imply always**(B) is transformed to $\{(\#A = 0) \vee (\#B > 0)\}$. [18] shows the rules generating an assertion from an IDL sentence.

Assuming that every loop is terminated (we assumed that a program is correct and has only finite execution paths), most of the assertions can be proved by Hoare logic. From the precondition of a program, we apply the axiom of each program statement and prove that an assertion is true. If one wants to prove the termination of a loop, Dijkstra's weakest pre-condition or a terminate function [8] can be used. To prove interprocedural information, one has to take care of parameters, return values and global variables. The detailed procedure to prove user information and an example of program verification are described in [18].

6. Path Processing

6.1. Intersection Through Information Terms

As we discussed in Section 4.3, the set of information paths translated from IDL information consists of a few patterns of regular expressions basically having only one statement label. Paths are constructed by connecting them with logical operators and encapsulating them with a scope closure. We call these patterns *information term* or simply *term* (t_I). Four types of information terms may appear in any translated information paths. We name each pattern as follows:

$(*a*)$	: ALLPATHTHRU
$\neg(*a*)$	: ALLPATHNOTTHRU
$\_a.\$s\ a.\$c\ (\_a.\$c)^k\ a.\$t\_$	: LOOPBOUNDS
$(\_a\_)^k$	: EXECCOUNTS

The first step in path processing is to simplify information paths. We first remove all negation operators. Using Demorgan's law recursively, $\neg$ operators in a path expression are moved down to the terms and finally combined with an ALLPATHTHRU term to be an ALLPATHNOTTHRU or vice versa.[3] Next, we transform it to a disjunctive form. Because of a scope closure, it is done at the two levels: inside and outside a scope closure.

Let σ_I denote a regular expression for the set of information paths translated form I. (We call it *information expression*.) For I_L and I_{S_i}, an IDL line (INFO_LINE) and a sentence (INFO_SENTENCE), respectively,

$$\sigma_{I_L} = \sigma_{I_{S_1}} + \sigma_{I_{S_2}} \ldots + \sigma_{I_{S_k}}$$

For each information sentence, its information expression is composed of a term of the scope and a scope closure as follows (see Table 2):

$$\sigma_{I_{S_i}} = \neg(*S*) + (\_S.\$s\ \sigma_{I_C}\ S.\$t\_)^*$$

where S is the scope of the sentence, and I_C is its information clause.

Inside a scope closure, we can transform σ_{I_C} to a disjunctive from. That is,

$$\sigma_{I_C} = (t_{11} \cap t_{12} \cap \ldots t_{1n_1}) + (t_{21} \cap t_{22} \cap \ldots t_{2n_2}) + \ldots + (t_{m1} \cap t_{m2} \cap \ldots t_{mn_m})$$

where t_{ij} is an information term.

Our strategy of path intersection is *localized modification*. First, intersection is achieved by modification. For $A \cap \sigma$ with any regular expressions A and σ, we resolve intersection by modifying A to be included into σ. More precisely, we remove all paths in A which are not included in σ, but no path is modified if it is in σ.

For a regular expression r, let $L(r)$ be the set denoted by r. Since we modify A (say A') such that

$$L(A') = L(A) - \overline{L(\sigma)}$$

where $\overline{L(\sigma)}$ means the complement set of $L(\sigma)$, and

$$L(A) - \overline{L(\sigma)} = L(A) \cap L(\sigma) = L(A \cap \sigma)$$

we have

$$A' = A \cap \sigma$$

Therefore, the modification resolves intersection.

The idea of localization is derived from the following observation. Suppose σ does not contain s or t (one may consider s and t special positioning symbols). Then,

$$A \cap (\_s\ \sigma\ t\_)$$

$$= \begin{cases} A_0\ s\ (A_1 \cap \sigma)\ t\ A_2 \text{ if } A = A_0 s A_1 t A_2, \text{ and none of } A_0,\ A_1 \text{ and } A_2 \text{ has } s \text{ or } t \\ \emptyset \qquad\qquad\qquad\qquad \textit{otherwise} \end{cases}$$

That is, special symbols localize intersection with an expression to intersection with its subexpression.

Suppose A is a set of program paths, and σ is a set of information paths. Since σ usually has constraints only on some subpaths within a scope, testing if a path in A is included in σ or not can be localized, and so can modification. Let $\sigma = (\_S.\$s\ \sigma'\ S.\$t\_)^*$. Then, from the above observation, $A \cap \sigma$ is resolved by locating every occurrence of scope S in A, and intersecting σ' with its subexpression enclosed by $S.\$s$ and $S.\$t$. In other words, $A \cap \sigma$ is eventually solved by localized interactions $\alpha_i \cap \sigma'$ where $S.\$s\ \alpha_i\ S.\$t$ is a subexpression of A.

One important aspect of localization is that a scope closure can be ignored in intersection. A scope closure in an information expression is essential to locate a subexpression of a program expression to be intersected, but it does not participate in actual intersection. As a result, assuming that localization by a scope closure is enforced implicitly, we can view an information expression from an IDL line, σ_{I_L} as a disjunctive form of information terms.

From the path model (Section 4.3),

$$\begin{aligned} X'_P &= A_P \cap I_P \\ &= A_P \cap (\sigma^1_{I_L} \cap \sigma^2_{I_L} \ldots \cap \sigma^n_{I_L}) \\ &= (\ldots((A_P \cap \sigma^1_{I_L}) \cap \sigma^2_{I_L}) \ldots \cap \sigma^n_{I_L}) \end{aligned}$$

where n is the number of information lines. Suppose $\sigma^i_{I_L} = (t^i_{11} \cap t^i_{12} \cap \ldots t^i_{1n_1}) + \ldots + (t^i_{m1} \cap t^i_{m2} \ldots \cap t^i_{mn_m})$ in a disjunctive form as above, then

$$\begin{aligned} & A'_P \cap \sigma^i_{I_L} \\ &= A'_P \cap ((t^i_{11} \cap t^i_{12} \cap \ldots t^i_{1n_1}) + \ldots + (t^i_{m1} \cap t^i_{m2} \ldots \cap t^i_{mn_m})) \\ &= (\ldots((A'_P \cap t^i_{11}) \cap t^i_{12}) \ldots \cap t^i_{1n_1}) + \ldots + (\ldots((A'_P \cap t^i_{m1}) \cap t^i_{m2}) \ldots \cap t^i_{mn_m}) \end{aligned}$$

where A'_P denotes an intermediate result of intersection with $\sigma^j_{I_L}$ for $j < i$.

From the above formula, we can see that path analysis is eventually a sequence of intersections with information terms. The algorithm intersecting four types of information terms is shown in Figure 8. Here, Π is a subexpression localized by a scope, which will be modified depending on the meaning of a term. Operator $\diamond$ will be explained in the next subsection.

Since path group information only summarizes program behavior, it is processed after path processing is finished. Processing path group information is again done by intersection. For each path group information, the regular expression translated from its condition clause is interseced with X'_P, the result of path processing. Since the intersection gives a set of program paths which are possibly executable and which satisfy the condition, it represents the corresponding path group. After computing the time bounds of the path set, we keep them with the given path group name in the procedure time table for reference in other procedures. (In fact, we also prepare the time bounds of the complement group for a given path group because a path group can be negated.)

6.2. *Complex Intersection*

Although information expressions can have special structures by introducing IDL, its intersection is not always easy. The complex case is when a statement in a loop is constrained. For example, when an information term is ALLPATHTHRU(a) and a is inside a loop (e.g., $(a + a')^K \cap (*a*)$), the intersection result includes every path passing through a at any iteration of the loop. It is already known that without an intersection operator, this type

```
intersect_term(Π, t_I(a))
    /*  Π is a regular expression localized by a scope
    t_I(a) is an information term on statement a     */
{
    case (t_I(a)) {
    ALLPATHNOTTHRU:
        every a in Π is replaced by ∅
        simplify Π by removing ∅
    LOOPBOUND:
        for every a in Π,
            replace closure star * with iteration with bounds (K) as given in t_I
    ALLPATHTHRU:
        if a is not in Π then Π ← ∅
        if a is inside a loop then Π ← Π ◇ t_I
        if a exists but not inside a loop then
            replace all alternative subpaths with ∅
            simplify Π by removing ∅
    EXECCOUNTS:
        Π ← Π ◇ t_I
    }
}
```

Figure 8. Algorithm of intersecting terms.

of path set requires an exponential length description enumerating all cases. Thus, we take special care of this *complex intersection*; resolve it no further in path processing and handle it in time computation. If $\Pi \cap t_I$ is complex, it is marked as $\Pi \diamond t_I$, but treated as $\Pi \cap t_I$ in the remaining path intersection. Then $T(\Pi \diamond t_I)$ is estimated during time computation.

The method to estimate the times of complex intersection is as follows. Let p_{best} and p_{worst} be the best and worst case path of $\Pi \diamond t_I$, respectively, such that

$$p_{best}, p_{worst} \in \Pi \diamond t_I$$

and for any path $p \in \Pi \diamond t_I$,

$$low(T(p_{best})) \leq low(T(p)) \text{ and } up(T(p)) \leq up(T(p_{worst}))$$

We first build p_{best} and p_{worst}, and then $T(\Pi \diamond t_I)$ is estimated as $[low(T(p_{best})), up(T(p_{worst}))]$. The prediction is safe, because

$$low(T(p_{best})) \leq low(T(p)) \leq t^b(p) \leq t^w(p) \leq up(T(p)) \leq up(T(p_{worst}))$$

where $t^b(p)$ and $t^w(p)$ means the best and worst execution times of path p.

In building the best path p_{best} (the worst case is handled in the same way), Π is used as a basis (template) of path construction, and t_I works as a constraint accepting the constructed path. We start from the best case path π_{best} of Π and modify it to be included into t_I, which we call *path modification*. As long as a modified path is kept in Π and in

the best case, it becomes the best path of $\Pi \cap t_I$. To be in Π, a path is modified only to another path in Π. To keep it in the best case, we minimize modification; since we started from the best case path, the less we change it, the closer it is to the best case path. Modification is done by unwinding a loop, replacing the subpath inside the unwound loop with the best alternative subpath defined both in Π and t_I, and arranging (or expanding) the loop bounds until t_I is satisfied. (There may be more than one best and worst path; p_{best} and p_{worst} become one of them.)

This path modification works well when there is only one complex intersection in a scope; however, it does not if a scope has multiple complex intersections (e.g., $\pi \diamond t_I^1 \diamond t_I^2$ or $(\pi_1 \cdot (\pi_2 \diamond t_I^1)) \diamond t_I^2$). Because of the relations among the information terms, it is difficult to keep a modified path in the best and worst case. There may be some ways, probably complex, to solve this problem, but we are adopting an easy approximation method here.

Our approximation method is to select the most effective intersection, meaning the intersection with a term which gives the tightest predictions. For $\Pi \diamond t_I^1 \diamond t_I^2 \ldots \diamond t_P^n$ (t_P^i can be on the different scopes inside Π), we apply a single complex intersection for each t_P^i and select the best and worst time among the results. That is,

$$T(\Pi \diamond t_I^1 \diamond t_I^2 \ldots \diamond t_P^n) \approx \overset{B}{\bigcap} T(\Pi \diamond t_I^i)$$

where interval-bound operator $\overset{B}{\cap}$ is defined as follows: $[a, b] \overset{B}{\cap} [c, d] \equiv [max(a, c), min(b, d)]$[4]. For any t_P^i,

$$(\Pi \diamond t_I^1 \diamond t_I^2 \ldots \diamond t_P^n) \subseteq (\Pi \diamond t_I^i)$$
$$T(\Pi \diamond t_I^1 \diamond t_I^2 \ldots \diamond t_P^n) \overset{B}{\subseteq} T(\Pi \diamond t_I^i)$$

Hence, the approximation is always safe.

6.3. Time Computation

Path analysis gives the set of possibly executable paths as a sum of subsets (i.e., $X_P' = \Pi_P^1 + \ldots + \Pi_P^n$). We focus here on how to use this result of path analysis and to compute a tighter prediction.

Several methods were considered. One of them is *program transformation* where for each subset Π_P^i, the original program (P) is transformed into a new program (P^i) whose program paths (A_{P^i}) are equal to Π_P^i. We can apply our prediction technique to each transformed program as it is, and compute $T(X_P')$ by merging the predictions of each program as explained in the path model. The benefit of this method is that path analysis is completely transparent to timing prediction. However, the method is likely to be inefficient because of transformation cost and repeated timing prediction. Temporally equivalent transformation including control cost seems also nontrivial, especially if a program has complex structures.

Conditional timing schema is another. The timing schema for a statement is expanded to a conditional formula to compute different times case by case. Each subset has its own timer and counts the corresponding time bounds for a statement, depending on whether it has the statement or not (i.e., whether its path passes through the statement). For example, given **if** statement S : **if** (B) **then** S_t **else** S_e,

$$T(S, \Pi_P^i) = \begin{cases} T(B) + T(S_t) + T(then) & \text{if } S_t \in^5 \Pi_P^i \wedge S_e \notin \Pi_P^i \\ T(B) + T(S_e) + T(else) & \text{if } S_e \in \Pi_P^i \wedge S_t \notin \Pi_P^i \\ T(S) & \text{otherwise} \end{cases}$$

where $T(S)$ is the old timing schema covering both cases together. With slight expansion of timing schema, we can use the result of program analysis directly in the same timing prediction method. Also it is more efficient than program transformation, because a program is read only once and multiple timers are updated together. However, the method suffers some complication that timers ($T(\Pi_P^i)$) should be managed dynamically.

Both methods provide clean separation for two tasks; path analysis works as an independent preprocessing, and the same or slightly modified timing computation technique can be used. However, regardless of implementation complexity and cost, neither method supports path modification with respect to time, which is important in handling complex intersections (Section 6.2).

The method we chose involves a *timed label*, where a label of a statement is expanded to have not only the identifier of a statement but also the execution times of its components. The execution times of each statement are predicted and recorded into its label by the basic schema-based prediction method, when the set of all program paths (A_P) is generated from a program text. After path analysis, the execution times for the set of possibly executable program paths (X_P') is computed using the times recorded in the labels.

In building timed labels, all of the timing information to be used later in time computation should be prepared. For a simple statement labeled by a user or default naming, its execution times are predicted and recorded into its label. If a statement is a labeled but nonsimple statement, its time bounds are not determined here because it may be manipulated in path processing. Instead, all timing components including control costs are prepared. Some of the components whose times are not predictable (e.g., nested nonsimple statements) keep a pointer to the corresponding subpath instead of time values. In case a statement is unlabeled, its times are determined and added to the times of a labeled statement that is in a straight-line with it in execution. This is always possible because there exists at least one label given by a user, the procedure name.

Since each label has its execution times and the timing of each operator[6] is also well defined, we can easily compute the time bounds of any subpath of a program at any time, even in the middle of path processing. This makes it easy to process a complex intersection, which requires a lot of path modification based on its times. The method is also as efficient as conditional timing schema. The only drawback is that it imposes conceptually more sophisticated combination than the above two methods; instead of two purely separate processing steps, timing prediction is divided into two substeps, and path analysis is activated in the middle of them.

The complexity of our timing prediction is purely that of path processing, because it starts with information analyzed by a user. All the basic steps in path processing: simple intersection with an information term, path modification for complex intersection, and time computation for each subset of paths (Π^i_P), are linear to the length of a program. It is obvious that the number of information terms is determined by the amount of user information provided. The number of the subsets also depends on user information. Theoretically, it can be as many as the number of program paths; consider the case when a user specifies every program path with IDL statements. However, this may not happen in practice (in the experiment, less than 10 IDL statements specifying less than 5 subsets for most procedures). In summary, considering the size of a program constant, the prediction cost is a function of the amount of user information, basically the number of IDL statements. Unless the amount is abnormally large, prediction can be made efficiently.

7. Experiments

We have built a prototype tool to predict the execution times of a program with the path analysis technique based on the path model. The tool inputs a C subset program and user information written in IDL, and outputs the behavior and the execution times of the program: the patterns of program paths and their times, the times of user defined path groups, and the execution time bounds covering all executions. The target system environment is the same as for the earlier timing tool: GNU C compiler and a 68010-based standalone SUN2.

Figure 9 shows the organization of the timing tool. The tool consists of 4 components, written in about 8,000 lines of C, YACC and LEX code. *Timed Ap-Generator* inputs a program text (P) and generates a regular expression (A_P) for the set of all statically possible program paths of P. While generating A_P, it also predicts the times of each statement and records them into the corresponding label in A_P as explained in Section 6.3. In *Ip-Generator*, the sets of information paths (I_P) are generated. Given IDL statements, Ip-Generator translates them into regular expressions line by line, and simplifies them as a disjunctional form of information terms.

Path-Processor eliminates the impossible paths in A_P by performing intersection between A_P and I_P. It continuously copies and modifies A_P as specified in I_P. The result is the set of possibly executable program paths (X'_P) decomposed into the sum of multiple subsets. Each subset consists of labels, +, ., $\hat{}$ (finite iteration) and $\diamond$ for complex intersection. *Time-Computation* computes the times of each subset in X'_P, and eventually the time bounds of X'_P as an estimation of $T(P)$. While computing times, complex intersection is handled by path manipulation, if it exists.

Figure 10 shows some part of outputs using the tool for the simple procedure "check_data." The program and user information are as in Figure 5 (except that the label LB for the loop body is not given explicitly). In the figure, $\hat{}$ [], ; and && mean operator *, $\cdot$ and $\diamond$, respectively. The tool analyzed the behavior of the procedure into two patterns of paths, the exception case (PATH1) and the normal case (PATH2), and gave their times. Since the user information also defined a path group EXCEPTCASE passing statement C (Figure 5), the tool gave the times of the path group, which is the same as PATH1 in this case. Finally, the execution time bounds of "check_data" on all cases were given by merging the times of both cases.

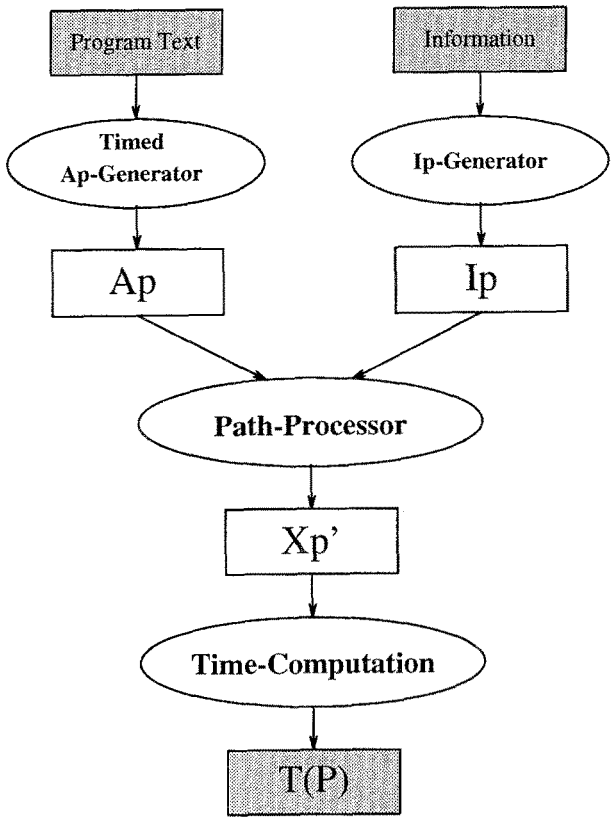


Figure 9. Organization of timing tool.

```
All Paths of 'check_data' :
   L ( ( A + B )  )^ []; ( C + C' )

PATH 1 :
  L ( ( ( A + B )  )^ [1,10] && ( _ A _ )^ 1  ) ; C
Time of PATH 1 : [ 666 , 2284 ]

PATH 2 :
  L ( B )^ 10; C'
Time of PATH 2 : [ 1886 , 2006 ]

Time of pathgroup EXCEPTCASE = [ 666 , 2284 ]

*** Procedure( check_data )
    Cycles = [ 666 , 2284 ]
    Times  = [ 67.75  ,  210.98 ] (micro-sec)
```

Figure 10. Example of running the tool.

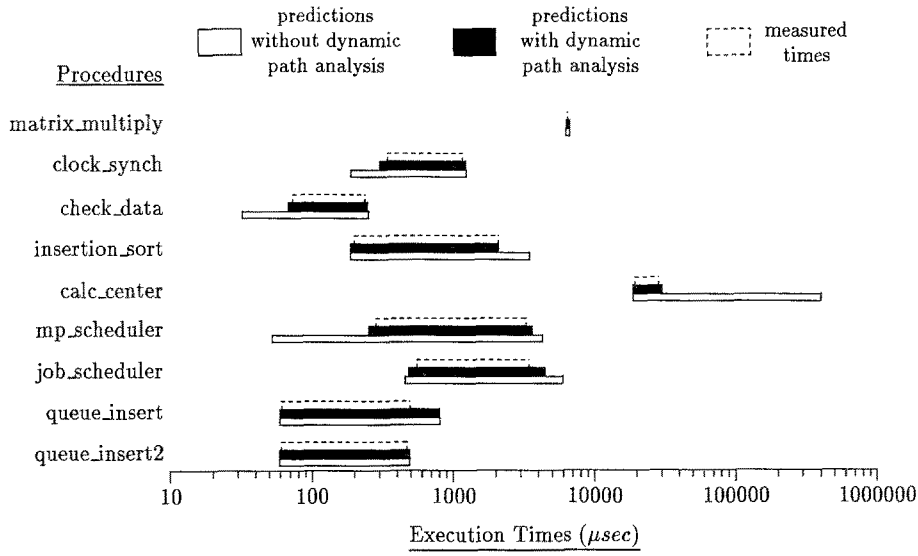
To validate our idea, we compared predictions with measurements as we did in [17]. We have tested a range of programs including a real-time kernel [2] and some famous algorithms. Figure 11 shows the time results of some interesting procedures. Each column in the table stands for predictions without dynamic path analysis,[7] predictions with dynamic path analysis and measured times.[8] Some examples are a single procedure (e.g., insert_sort), and others consist of multiple procedures (e.g., mp_scheduler consists of 5 procedures and 140 source lines).

For all procedures, predictions are safe regardless of dynamic path analysis, covering the measured times for the best and worst case. Predictions with static path analysis only are tight for some procedures (e.g., matrix_mutiply), but looser for some other procedures (e.g., calc_center); especially the upper bound of calc_center is more than 10 times greater than the worst execution time. With the help of dynamic path analysis, predictions become tighter. Counting interference from memory refresh, most of the predictions are very close to the measured times. The exceptional cases are job_scheduler and queue_insert.

For job_scheduler, dynamic path analysis gives much tighter bounds, but they are still loose. The given IDL information generates multiple complex intersections, and our approximation for them (Section 6.2) does not work as well. In case of queue_insert, the expressiveness of IDL causes a problem. The procedure has two loops in sequence, and there exists an interesting relation between them: each loop can iterate [0, #_of_processes] times but the sum of the iterations of the two loops is also in [0 , #_of_processes][9]. (This type of loop relation was mentioned as *loop sequence* in [19].) This information is describable in our path language, regular expressions (i.e., $(L1 + L2)^K$), but cannot be expressed in our subclass interface language, IDL. Path analysis can do nothing because no information can be provided.

One interesting issue is timing predictability of a procedure. We could classify procedures into three types as follows. The first type is a procedure whose execution times are predictable tightly without dynamic path analysis. This type procedure usually performs some confined operations. It is interesting that more than half of the tested procedures fall in this type. A procedure is the second type if its predictions using only simple timing schema are loose, but they can be refined tight by path analysis using IDL information. We could find this type in many small algorithms (e.g., sorting) and some high level procedures that consist of many procedure calls. This type is less frequent than the first type, but it is usually critical in determining the execution times of a program. The last type is a procedure whose predictions are loose even with path analysis. A procedure may need complex information that cannot be described in IDL (e.g., queue_insert), or that is describable but not sufficiently processable (job_scheduler). We expect that this type of a procedure is not often found.[10]

Certainly, predictability depends on what a program does and which algorithm it implements. However, it is also affected by programming style. For example, we could write queue_insert2, which is the same procedure as queue_insert except that the two loops are combined into one loop. As shown in Figure 11, queue_insert2 is much more amenable to prediction of execution times; no complex information is required for very tight bounds. (This experiment shows that a timing tool can be used as an aid to write a predictable and efficient program with respect to performance.)



procedure	predictions [a] without dynamic path analysis	predictions with dynamic path analysis	measured times (μsec)
matrix_multiply [b]	[6276.82 , 6696.40]	[6276.82 , 6696.40]	6412
clock_synch [c]	[186.97 , 1221.96]	[298.66 , 1221.96]	[338.5 , 1153.9]
check_data [d]	[32.15, 252.00]	[67.75 , 248.60]	[73.1 , 234.9]
insertion_sort [e]	[187.17, 3450.10]	[187.17 , 2085.04]	[197.2, 2071.1]
calc_center [f]	[18685.00 , 400828.43]	[18685.00 , 30338.89]	[19313 , 28476]
mp_scheduler [g]	[51.88, 4253.31]	[248.21, 3618.12]	[280.7 , 3242.6]
job_scheduler [h]	[454.71 , 5950.37]	[477.29 , 4480.08]	[548.6 , 3444.0]
queue_insert [i]	[58.80 , 798.51]	[58.80 , 798.51]	[60.5 , 490.4]
queue_insert2 [j]	[58.80 , 489.56]	[58.80 , 489.56]	[60.5 , 469.3]

[a] All predictions and measured times include [0,7]% (average 5%) delay caused by nondeterministic interference from memory refresh [17]

[b] The matrix size is 5×5.

[c] Perform averaging calculation of clocks from at most 8 sites.

[d] The same program in Figure 5 with "handle_exception" taking [24.01 , 25.84] μsec.

[e] The number of data elements sorted is 10.

[f] Calculate the center of an object image (a down-scaled procedure of the example in [19] There is an assumption that the object has a restricted size.

[g] Multiprocessor scheduler that allocates 5 processors.

[h] Schedule 10 jobs in earliest deadline first.

[i] Insert a process into a two-level priority queue.

[j] A rewritten procedure of "queue_insert".

Figure 11. Time results of sample procedures.

It is worthwhile to note that partial information may be enough to predict sufficiently tight bounds; complete knowledge of program paths is not always necessary. For the sample procedure mp_scheduler, we could find 12 IDL information statements. Experiments

showed that only half of them are effective to tighten predictions; the other half give minor gain, or have no effect after approximation. This observation tells us that not all information statements have the same value with respect to timing prediction, and that part of them only can refine predictions tightly enough. Therefore, it is a good strategy for a user to give information incrementally: give obvious and large-scale information first, and add more detail if predictions are not sufficiently tight. It can reduce not only unnecessary processing cost but also overhead to verify given information.

There is no definite answer to the question about which type of information is more valuable. It all depends on the static structure and the dynamic behavior of a program. Our empirical observation is as follows. Information on loop iteration number is necessary; we cannot predict the times without it. Execution count is usually decisive, if it exists (e.g., calc_center). The complexity results of an algorithm are usually expressed with it. Path relations, including interprocedure relations, are useful when a program performs some complicated operations with multiple cases (e.g., mp_scheduler).

The results of our experiments show the following points. All the predicted times are safe. Frequently, tight predictions can be obtained by our efficient static path analysis only. For most procedures, predictions with dynamic path analysis are fairly tight to the best and worst case execution times. Complete knowledge may be not necessary, because some partial information can result in sufficiently accurate predictions. There exist some cases in which our path analysis with IDL is not sufficiently helpful. These problematic cases are not the limit of our path model, but the result of our decisions for a practical solution through IDL, expecting that those cases are ineffectively rare. Some of them can be handled by rewriting a program in a more predictable way.

8. Future Work

While designing the interface language, the primary concern was processibility with respect to path intersection and time computation. We started from a general one which allows a sequence of statements to be a basic object, which is as descriptive as regular expressions. We continued by trial and error to add or delete restrictions, and finally decided upon IDL. We believe that IDL is very restrictive, maybe close to the minimal in the sense that if another restriction is added, it cannot describe some useful user information. There is always a chance to expand IDL and process more powerful user information (e.g., an information clause can be placed on a condition clause). However, it usually costs more complex processing; even IDL is not so easy that we need to make an approximation in time computation. An expansion should be made when it causes no or little increase of processing complexity, or when experiments tell it necessary for valuable user information. One may also define other alternative types of interface languages, but it should be easy to use, process and verify.

Although we are confident that path processing is correct and effective giving safe and very tight predictions, there are some questions yet to be answered. How many real programs need dynamic path analysis to get satisfactory predictions? Is IDL easy to use and sufficiently descriptive? Is IDL too restrictive to describe some user information? Ease

of use is somewhat subjective, but it certainly depends on what kind of execution information a user has. Usefulness of path analysis may be related with several factors including the complexity of a program, the accuracy of basic prediction method, and the degree of precision requested. A lot of experimental work, especially with programs used in real systems, should follow to address these problems.

As a refinement method, our path analysis works well for sequential programs. An emerging question is whether a similar technique works for concurrent programs too. We have developed timing schema for concurrent constructs such as locking and message passing [21], and initial experiments showed that timing prediction of a parallel program is feasible with those timing schema [9]. Although this simple approach gives safe predictions, it has the same problem of loose predictions counting impossible cases as most static analysis methods do. We are investigating a way to apply the same framework of compensation based on user information. User information should be expanded to include not only paths for statement relations in a process but also statement and temporal relations with other processes, especially blocking times (e.g., Process A and B do not request a resource at the same time). The key issue is again how to generalize execution information.

Program behavior analysis based on the path model may be used in some areas other than timing prediction. For example, it may help test data generation. Generating test data usually suffers from an exponential path domain and frequent backtracking because many of the paths are revealed as infeasible in the middle of data generation [10]. Eliminating infeasible paths can reduce many fruitless efforts. Certainly, our path analysis cannot tell the correctness of a program. However, it may be used to check quickly whether a program is compatible with some desired program behavior. By performing path analysis with IDL descriptions of the desired behavior, we can see whether it is defined in a program and how other program behavior reacts with it. Path analysis may also supply some runtime information. With possible behavior decomposed into several cases, one may predict future behavior in an early stage of execution. This knowledge may be useful in many areas such as efficient runtime resource management.

Recently, several studies have mentioned an approach of *runtime prediction* (e.g., [7]). Their idea is to express the execution times of a program as a function of the program variables at compile time, and to evaluate the function at runtime knowing the values of the variables. It can avoid a pessimistic prediction and thus provide a possibility of better dynamic decisions, such as the guarantee of a dynamic request from a process. However, the approach implies additional complexity in compile time analysis (e.g., symbolic computation) and overhead to compute a function at runtime. Our pathwise analysis may be the better solution for this purpose. We can prepare predictions for each possible execution case of a program. Then at runtime we check which case is actually being executed and select the tight predictions dynamically among the predictions prepared. The only overhead is cost for recording the execution history information on some significant statements, instead of monitoring the values of the variables in the former approach.

Input/output operations are particularly important in real-time systems because they represent interactions between a system and the environment, and take a major part in system operations. We modeled an I/O operation as execution of an I/O statement, and applied the same schema-based static analysis [18]. The main problem in I/O analysis is predicting waiting time for a device or data and interference caused by asynchronous execution

in a variety of implementation policies. Our approach was to develop a general framework and to apply it to a specific case. We developed refinement rules for possible implementation policies that transform an implementation-independent timing schema to implementation-dependent, and thus more predictable, timing schema. We also introduced interference formulas that estimate the effect of I/O interference on program execution times for given implementation policies of a target system. We showed several examples of analysis applying the framework to specific cases.

With a program logic extended with a real-time clock, our performance predictions can reason about other timing properties of a system [20]. Although our predictions are made on the assumption of maximum parallelism, they can be applied directly in some systems where a processor is shared in a predefined way (e.g., cyclic executives). Processor sharing with a special task (e.g., interrupt handler) can also be handled easily by slightly modifying predictions for interference from the task. For general processor sharing, one may depend on scheduling theory. Our predictions are projected to the computation time of a task, and system timings (basically, deadlines) are determined by checking the feasibility of the task set. The problem here is how realistic a task model is. The approach we are considering now is to expand our interference model for general processor sharing. We consider all delays caused by processor sharing interference, and adjust predictions for those delays. Starting from a simple environment and adjusting predictions step by step, we expect to have predictions in a real environment.

In this paper, we introduced an idea combining dynamic path analysis with a timing prediction method. The prediction technique is a simple and efficient static analysis based on timing schema. Using powerful information provided by a user, dynamic program analysis eliminates the effects of infeasible paths and refines predictions tight. As a basis to exploit user information for dynamic path analysis, we developed the formal path model. Its practical application was achieved by introducing a user interface language IDL. Experiments with a timing tool showed that our approach is valid and promising; safe and tight execution time predictions are possible for a wide range of programs in an effective way.

Acknowledgment

I am grateful to Becky Callison, Travis Craig, Ricardo Pincheira, Sitaram Raju for their helpful comments and suggestions. Special thanks to Becky Callison and Sitaram Raju for their very careful reading of this paper. I would like to thank Professor Alan Shaw for advising me throughout this work with his insight and experience.

Notes

1. A simple statement is a statement that does not contain any other statement (e.g., an assignment statement).
2. Here, **exclusive** means not *mutually exclusive* but *exclusive or*.
3. From the syntax of IDL, a negation operator comes only with ALLPATHTHRU or ALLPATHNOTTHRU.
4. We assume that two interval bounds are overlapping, i.e. $(a \leq d) \wedge (c \leq b)$.
5. Here, $s \in \Pi$ means $\exists p \in \Pi$ such that p passes through s.

6. The timing of each regular expression operator is defined as follows:

$$T(a \cdot b) = T(a) + T(b),$$

$$T(a^K) = K \times T(a), \text{ and}$$

$$T(a + b) = T(a) \overset{B}{\cup} T(b) \equiv [\ min(\ low(T(a)), low(T(b))\ ),\ max(\ up(T(a)),\ up(T(b))\ )\ ]$$

7. User input for loop bounds are needed also in the early version tools based on timing schema only [17].
8. We generated a data set for each of program paths that can possibly be the best and worst case with respect to execution time, and measured the time of 150,000 iterations (15,000 for some procedures) of execution for each case. The bounds of measured times were computed from the shortest and longest measurements.
9. The first loop searches for an insertion point based on the first-level priority. If there exists a tie, the second loop decides the proper position based on the second-level priority.
10. In the experiment, we have tested more than 50 programs, and those two are the only examples.

References

1. Avrunin, G., Dillon, L., Wileden, J., and Riddle, W. 1986. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*. 12:278–291.
2. Callison, H., and Shaw, A. 1991. Building a real-time kernel: First step in validating a pure process/Adt model. *Software — Practice and Experience*. 21:337–354.
3. Chen, M. 1987. TAL — A language for timing analysis. Department of Computer Science, University of Texas, Austin.
4. Dijkstra, E. 1976. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall.
5. Hoare, C. 1969. An axiomatic basis for computer programming. *Communications of ACM*. 12:576–580.
6. Huang, J. 1990. State constraints and pathwise decomposition of programs. *IEEE Transactions on Software Engineering*. 16:880–896.
7. Gehani, N. and Ramamritham, K. 1991. Real-time concurrent C: A language for programming dynamic real-time systems. *The Journal of Real-Time Systems*. 3:377–405.
8. Gries, D. 1981. *The Science of Programming*. Berlin/New York: Springer-Verlag. Chapter 12, pp. 149–162.
9. Kim, J. and Shaw, A. 1990. An experiment on predicting and measuring the deterministic execution times of parallel programs on a multiprocessor. Tech. Report #90-09-01, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, WA.
10. Korel, R. 1990. Automated software test data generation. *IEEE Transactions on Software Engineering*. 16:870–879.
11. Lin, K., Kenny, K., Natarajan, S., and Liu, J. 1990. FLEX: A language for real-time systems programming. *Foundations of Real-Time Computing: Formal Specifications and Methods*. (ed. A. Tilborg and G. Koob), Kluwer Academic Publishers. pp. 251–290.
12. McNaughton, R. and Yamada, H. 1960. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*. 9:39–47.
13. Mok, A., Amerasinghe, P., Chen, M., and Tantisirivat, K. 1989. Evaluating tight execution time bounds of programs by annotations. *Proceedings of 6th IEEE Workshop on Real-Time Operating Systems and Software*. pp. 74–80.
14. Niehaus, M. 1991. Program representation and translation for predictable real-time systems. *Proceedings on 12th IEEE Real-Time Systems Symposium*, pp. 43–52.
15. Nirkhe, V. and Pugh, W. 1991. A partial evaluator for the Maruti hard real-time system. *Proceedings on 12th IEEE Real-Time Systems Symposium*, pp. 64–73.
16. Olender, K. and Osterweil, L. 1990. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*. 16:268–280.
17. Park, C. and Shaw, A. 1990. Experiments with a program timing tool based on source-level timing schema. *Proceedings on 11th IEEE Real-Time Systems Symposium*. pp. 72–81. (A revised version is also in *IEEE Computer*, 24:48–57.)

18. Park, C. 1992. Predicting deterministic execution times of real-time programs. Ph.D. Thesis, University of Washington, Department of Computer Science.
19. Puschner, P. and Koza, Ch. 1989. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*. 1:159–176.
20. Shaw, A. 1989. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*. 15:875–889.
21. Shaw, A. 1991. Deterministic timing schema for parallel programs. *Proceedings of 5th International Parallel Processing Symposium*. pp. 56–63.
22. Stoyenko, A. 1987. A real-time language with a schedulability analyzer. Ph.D. Thesis, Univ. of Toronto, Computer Systems Research Institute, Tech. Report CSRI-206, Toronto.
23. Stoyenko, A. and Marlowe, T. 1992. Polynomial-time transformation and schedulability analysis of parallel real-time programs with restricted resource contention. *Real-Time Systems*, 4:307–330.
24. Woodward, M., Hedley, D., and Hennell, M. 1980. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*. 6:278–286.
25. Young, M. and Taylor, R. 1988. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14:1499–1511.