TimeC: A Time Constraint Language for ILP Processor Compilation*

Allen Leung[1] Krishna V. Palem[2] Amir Pnueli[3]

[1] leunga@cs.nyu.edu
Courant Institute of Mathematical Sciences

[2] palem@cs.nyu.edu
Courant Institute of Mathematical Sciences

[3] amir@wisdom.weizmann.ac.il
The Weizmann Institute of Science

Abstract. Enabled by RISC technologies, low-cost commodity microprocessors are performing at ever increasing levels, significantly via *instruction level parallelism (ILP)*. This in turn increases the opportunities for their use in a variety of day-to-day applications ranging from the simple control of appliances such as microwave ovens, to sophisticated systems for cabin control in modern aircraft. Indeed, "embedded" applications such as these represent segments in the computer industry with great potential for growth. However, this growth is currently impeded by the lack of robust optimizing compiler technologies that support the assured, rapid and inexpensive prototyping of real-time software in the context of microprocessors with ILP. In this paper we describe a novel notation, TimeC, for specifying timing constraints in programs, *independent* of the base language being used to develop the embedded application; TimeC specifications are language independent and can be instrumented into imperative and object-oriented languages non-intrusively. As we will show, the program synthesis problem that arise out of Time_tract specifications, a subset of TimeC, are always "tractable". In contrast, a range of specification mechanisms proposed earlier yield substantially intractable synthesis questions, thereby limiting their potential utility. We will compare the tractability and related expressive power issues between TimeC and some of the extant mechanisms for specifying properties of timed programs.

Keywords: instruction scheduling, compiler optimizations, embedded systems, real-time systems, timing constraints.

*Supported in part by an award from Hewlett-Packard Corporation, IBM corporation, Panasonic AVC Labs, DARPA contract no. DABT63-96-C-0049, and an NYU Research Challenge Grant.

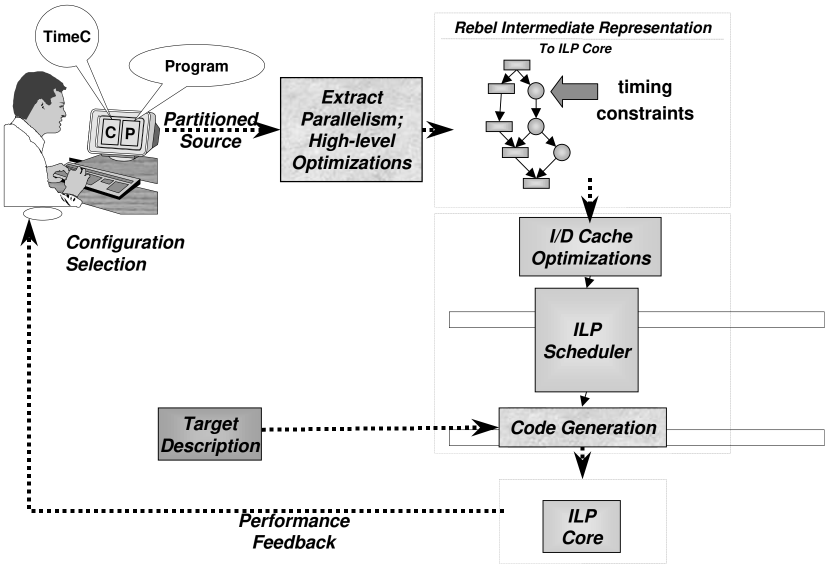


Figure 1: The TimeC development environment.

1 Introduction

In this paper we introduce a language independent notation, TimeC, for specifying timing constraints in *embedded programs*, which typically require specific timing relationships to hold, between individual instructions, as well as between groups of instructions. Multi-media servers providing images in a video on demand setting, automobile brake systems based on ABS technology and medical control applications are all examples of such embedded systems; we will consider a detailed example in Section 2.1. TimeC (shown in Section 2) is meant to be a guide to the compiler as well as the run-time environment, to help monitor and verify that timing constraints specified by the programmer are enforced.

Our view is that in the presence of an optimizing compiler as shown in Figure 1, TimeC is at the heart of an environment for generating executable code that meets the constraints automatically. Thus, we can view our time-constraint languages as providing the foundation of "synthesizing" embedded applications from high-level (imperative or OO language) specifications automatically.

The current art in developing embedded software involves a substantial prototyping effort since compilers and other programming tools do not readily support timing constraints. For example, a programmer would informally identify points of interest in the program, associate preferred timing relationships between these points and then determine by hand whether these relationships in fact hold. The process of specifying constraints as well as that of ascertaining that they hold when the program executes—often referred to as timing *analysis*—is done with minimal support from the compiler. By contrast, our notation will permit the compiler to help automate the specification as well as the analysis and synthesis phases.

Briefly, TimeC constraints can be specified between program points delineated by markers. For a marker M, we consider the *visit time* at M, denoted by $\mathsf{time}(M)$, to be the time at which the instruction immediately preceding M has just been completed. Informally, markers are associated with program points between instructions. Thus, given a pair of markers, TimeC constraints can be used to define bounds on the "interval" of time that elapses between them, in all executions of the program. For example, a constraint of the form

$$\mathsf{time}(\mathsf{next}.M_1) - \mathsf{time}(M_2) \leq 10$$

states that there can be no more than 10 units of delay between the *externally observable* events delimited by the markers M_1 and M_2 in the program.

TimeC constraints are *language independent* in that the embedded application program $\mathcal{P}$ can be specified in any extant imperative languages, such as the C programming language. This notation is then communicated to the front-end via a preprocessor that annotates $\mathcal{P}$ with the timing constraints, which in turn are propagated to the intermediate representation in the compiler.

As shown in Figure 1, a modern compiler utilizes many phases of optimization that alter the structure of a program and hence the timing relationships between parts of the program. A significant goal of our effort is to provide TimeC as a basis for constraining these optimizers further: *in addition to preserving conventional program semantics—these are typically characterized via control and data dependences—the next generation of optimizers could additionally preserve and propagate timing relationships as well.*

A significant issue of using TimeC constraints in the context of compiler optimizations is their impact on the complexity of the implementation. Currently, typical compiler optimizations are very "light-weight": they run fast and have relatively simple implementations. Adding timing constraints with languages such as TimeC can significantly increase the running times as well as the implementation complexity.

In this context, *instruction scheduling* is an optimization that plays a significant role. To understand this, we need to consider a modern microprocessor that typically permits the execution of multiple instructions on a single cycle. These microprocessors are increasingly allowing the compiler to control how these instructions are issued—notable examples included the *explicitly parallel instruction computers (EPIC)* that have been announced via a joint effort by the Hewlett Packard and Intel corporations [7]. In such processors, the instruction scheduling optimization rearranges the source program so that

- As many parallel instructions as possible are executed on each and every cycle
- while preserving the programs data- and control dependences.

Now, given timing constraints, instruction scheduling must also enforce them. Clearly, the complexity of the timing constraint language has a direct impact on the complexity of the algorithms used in scheduling.

Motivated by this concern, we also present in Section 4 a simpler form of the TimeC language which we refer to as Time_tract. Time_tract has the useful property that the complexity of instruction scheduling programs with these constraints is no greater than those of conventional instruction scheduling optimizations without timing constraints; we will show this in Section 6. In a companion paper [23], we describe algorithms for instruction scheduling enhanced by timing constraints.

In the context of compiling or synthesizing embedded applications, our work is the first proposal for specifying timing constraints in programs that:

1. is independent of the language in which the application is being developed,
2. is explicitly designed with concern for the complexity of the resulting compiler optimizations.

Most of the previous work on timing constraint formalisms, such as RTL[16] and Hoare Logic with time[29], concentrated on the tasks of specification and verification of real-time systems. Consequently, these formalisms are typically very expressive which makes the goal of automatic synthesis according to a given specification highly intractable. Currently, the only language notations that seriously consider the synthesize problem are TCEL[15] and CRL[31]. However, in contrast to TimeC, both are new programming languages in their own right, and the programmer can only utilize the full power of the timing constraints within these new languages. We will return to comparisons of TimeC and Time_tract with other work in Section 7.

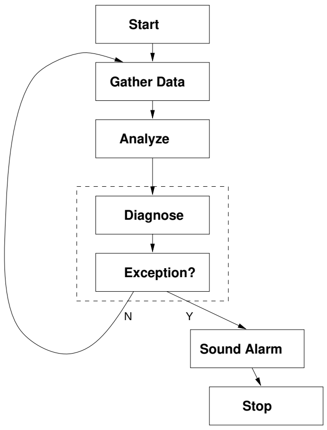
2 The TimeC specification language

As mentioned in the introduction, TimeC is intended to be a timing specification language suitable for the description of real-time constraints, with special emphasis on compile time issues. Unlike other real-time languages, TimeC only concerns itself with the specification of timing properties of an application; the functional semantics of an application are specified using a conventional imperative language such as C, Fortran, C++, etc. as the *base* language.

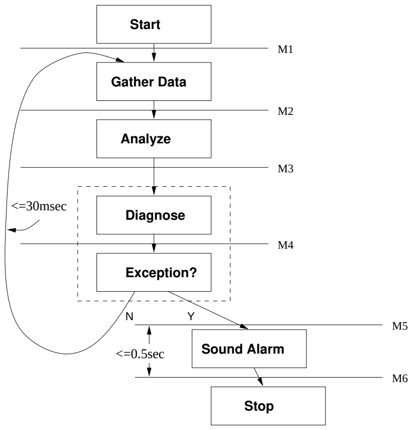
In this section we will describe the syntax and informal semantics of TimeC. The formal semantics of TimeC is described in section 3. We will start our discussion with an example. For convenience, we will first present the notions in the context of basic-blocks [1], [27].

2.1 An Example Embedded Program

Consider a program that controls a typical Cardiac Arrythmia monitor in a modern ICU [25]. In this case, we have a program $\mathcal{P}$ specified in an imperative or object-oriented language such as C or C++, as shown in figure 2(a) at the granularity of functional basic-blocks. The points of the program of interest are denoted by "markers" $M_1, M_2, \ldots, M_6$ (figure 2(b)), between which timing relationships are to be enforced. Markers are associated with "events" of interest; by convention, let us associate the event associated with marker M_i to be the beginning of the execution of the instruction immediately following it[1].



(a)



(b)

Figure 2: An Example Embedded Application Program

The two constraints of interest, stated informally here and comprehensively developed below require that:

1. Data is sampled every 30 msec (two consecutive sampling points are never more than 30 msec apart), and

2. Whenever an abnormality (arrythmia) is detected, software sounds an alarm within 500 msec.

Throughout this paper, we will assume that all our formal specifications refer to structured programs [1], and that all times are expressed in milliseconds.

2.2 Syntax of TimeC for basic-blocks

Continuing with the example, let us consider the basic-block enclosed by markers M_5 and M_6. The constraint that the alarm be activated within 500 msec of the detection of an abnormality conidtion can be easily captured

[1] We can accommodate alternate conventions which associate the execution of the instruction immediately preceding the marker to be its associated event.

as constraints on the times $\mathsf{time}(M_5)$ and $\mathsf{time}(M_6)$, which denote the times at which markers M_5 and M_6 were visited. For example, the constraints we need to specify in order to satisfy the second requirement is

$$\mathsf{time}(M_6) - \mathsf{time}(M_5) \leq 500.$$

Let us now describe the notation used in order to impose such TimeC constraints on a program in some base language of our choice.

2.3 Adding TimeC to a base language

In general, we assume that the notion of *control points* is well-defined in our base languages. For example, statements in the C language [10] are executed sequentially as required by its semantics. The control points in the C language are simply the locations in front of each statement. For completeness, we also introduce a control point at the end of the program.

In this paper, we only consider languages with a single thread of control. For example, imperative languages like Fortran, C, C++, and task-free subset of Ada all fall under this category. Due to their simplicity, both K&R C and ANSI-C are suitable *light-weight* TimeC base languages.

In a more complex language like C++[32], the notion of sequencing is more problematic and may not be apparent from the program text. For example, a declaration of a variable of class **C** may implicitly invoke *constructors*, which in turn may invoke other constructors from base classes and/or constructors for data members of the class. A symmetrical problem arise from data type *destructors*. At the end of a scope destructors for objects are implicitly invoked, which may in turn invoke other base class destructors. In general, we will consider these constructors/destructors invocations as “hidden” procedure calls, and will treat them accordingly. For technical reasons, we will also disallow the use of non-local control flow constructs such as setjmp/longjmp in C or throwing an exception in C++.

Thus without further complicating the discussion in the sequel, we will restrict our attention to only the following basic programming constructs: (i) assignment, (ii) sequencing, (iii) conditional branches (if/then/else), (iv) while loop with exits, and (v) function/procedure call and return. Other constructs can be considered to be derived forms. We also assume that all control constructs are nested and each control construct has only one single entry (but multiple exits are allows.) In particular, unrestricted goto statements are prohibited, and each procedure contains a single entry point. Most programs that do not adhere to these restrictions can be rewritten into our restricted form without much difficulty.

2.4 TimeC annotations and specifications

Suppose we choose some suitable language $\mathcal{L}$ as our base programming language. Given a program P in $\mathcal{L}$, we assume that P contains a set of event generating constructs. Typically these are input or output statements whose invocation or completion correspond to some sort of observable real-time events. In general, the constructs in any language $\mathcal{L}$ can be categorized into two classes: (i) *externally-observable* actions, and (ii) *internal* actions. For example, input and output actions that communicate with the external environment fall into the first class, while computation statements typically fall into the second class. In the example of figure 2, data gathering and alarm sounding are accomplished by I/O actions of the first class. The timing constraints that we impose on a program are mainly relevant with respect to statements and instructions that generate an externally-observable action. The set of *externally-observable* actions is parameterized by the base language $\mathcal{L}$ of our choice.

In this and the remaining sections we will use the C language as our example base language to illustrate the process of incorporating TimeC. We will assume that we have access to two I/O routines *input* and *output* that generate externally-observable events.

The TimeC language is divided into *annotations* and *specifications* as shown in figure 3. Annotations are declarations made within the source program, which we will describe in subsection 2.5 and 2.6 respectively.

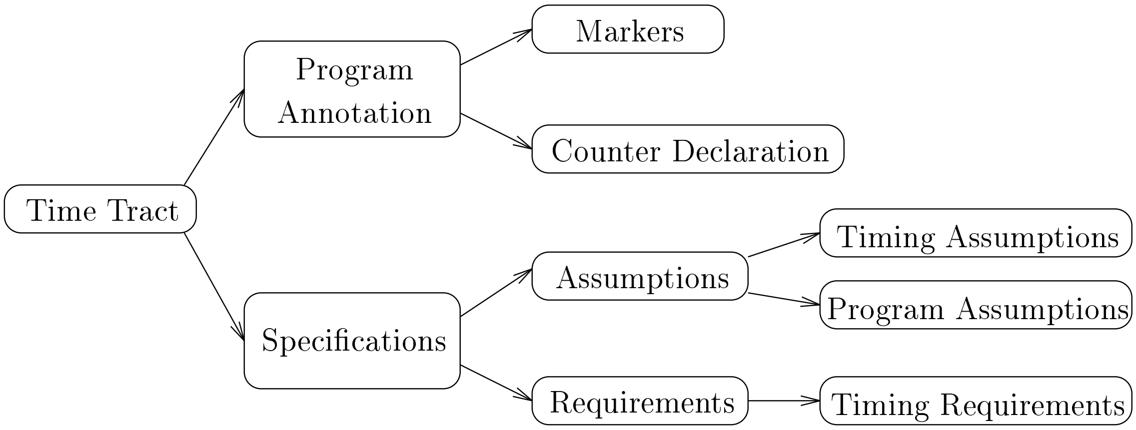


Figure 3: Structure of the TimeC language.

The simplest way in which a program can be annotated is through program *markers*. In order to avoid extending the syntax of the base language we have chosen to embed the markers inside the program comments. The symbol `!` is a special *escape* character that signifies the beginning of a TimeC annotation. For example, in the C language comments are enclosed within `/* */`. Thus, any C comments of the form `/*!...*/` are considered to be TimeC annotations.

Specifications, on the other hand, are presented outside the program text and these include *assumptions* and *requirements*. The assumptions inform the system about facts that cannot be automatically inferred from the program text, such as bounds on the execution time of activities that are outside the program (such as externally called procedures), and bounds on the number of iterations particular loops will be executed. Consequently, the class of assumptions is partitioned into *timing assumptions*, which specify bounds on the timing of external activities, and *program assumptions*, which specify bounds on variables and numbers of iterations. The *requirements* specify the timing relationships which should hold in a correct program and the system is responsible for their enforcement. For example, the timing behavior of system calls and I/O routines are typically non-deterministic. However, in a given application it is often possible for the programmer to provide lower- and upper-bounds on their execution times; for example, profiling can be used to obtain these data. From these assumptions provided by the programmer the system can extract extra information for scheduling and other optimizations, and attempt to ensure that the specifications are met by the scheduled program.

2.5 Annotation by Markers

In TimeC, timing constraints are always expressed in terms of time of events, and events are always associated with visits to certain sequence (control) points in the program. In order to be able to refer to the times of visits at these locations, we place *markers* at these points. A marker declaration consists of an identifier, the marker name, enclosed within the special `/*!...*/` annotation.

For example, the program of figure 4(a) is annotated with the three markers M_1, M_2, and M_3. Marker M_1 is associated with the start of the block while M_3 is associated with its completion. On the other hand, the marker M_2 is associated with completion of the output statement *output*(*chan2*,2) only. To signify that marker M_2 is visited only if the **else** branch is taken, we included its annotation before the semicolon terminating this statement but after the *output* procedure call.

Suppose we want to impose the conditions that the compound statement must complete within 10 msecs and that the **else** branch, if taken, must complete within 5 msecs. We can specify the timing constraints as in figure 4(a). These constraints state that the time differences between the visits to markers M_1 and M_3 and between the visits to markers M_1 and M_2 must be bounded by 10 and 5 msecs respectively.

There are two important clarifications that we should make here: (i) since the **if** statement may not always take the **else** branch during execution, the second constraint is vacuously true if the **else** branch is not taken and therefore marker M_2 is not visited; and (ii) The computation statement $c = x * x - y * y$ is free to be moved out of the compound statement by the scheduler if need arises— for example to satisfy the timing constraints. As mentioned in section 2.3, computation statements do not generate any externally observable events. Thus, performing code motion on these statements can produce a program with identical event ordering from the perspective an external observer, as long as the functional semantics of the program is not violated. In contrast, code motion that crosses marker boundaries in general cannot be applied to statements with externally-observable effects, since this may modify the relative ordering of externally observable events.

```
/*! M1 */
{
    c = x * x - y * y;
    if (c > 0) output(chan1,1);
    else output(chan2,2)
        /*! M2 */;
}
/*! M3 */
```

(a)

Requirements:

$\mathsf{time}(M_3) - \mathsf{time}(M_1) \leq 10$;
$\mathsf{time}(M_2) - \mathsf{time}(M_1) \leq 5$

(b)

Figure 4: An annotated C program.

TimeC marker annotations are simply comments, and they can be easily ignored if only the functional semantics of an application needs to be tested.

2.6 Multiple Visits to a Marker

In the absence of loops and procedure calls, every marker can be visited at most once, as we have seen in the previous example. In the presence of loops and procedure calls, however, a marker may be reached several times during the execution of a program. For example, in figure 2 marker M_2 is reached every time a new sample of data is gathered.

In general, every time constraint which refers to one or more markers is intended to be applied to each visit to this marker. Typically, a constraint refers to two markers and states a (lower or upper) bound about their temporal distance. In the case that one or both of the references markers are visited more than once, we usually identify one of them as the *reference marker* and the other as the *relative marker*. Relative markers are identified by having a *next* or *prev* prefix.

A constraint that refers to the reference marker M_1 and the relative marker *next*.M_2, should be applied to any instance of (visit to) M_1 and the *first* following instance of (visit to) M_2. Symmetrically, A constraint that refers to the reference marker M_2 and the relative marker *prev*.M_1, should be applied to any instance of (visit to) M_2 and the *most recent* preceding instance of (visit to) M_1.

Consider the annotated program in figure 5, consisting of an inner loop on j embedded within an outer loop on i. The inner loop contains two markers: start_inner and end_inner which straddle the body of the inner loop. In addition, the outer loop contains the single marker outer.

The specification part of this program consists of two constraints. The first constraint:

$$\mathsf{time}(\mathsf{next.end\_inner}) - \mathsf{time}(\mathsf{start\_inner}) \leq 2,$$

requires that the body of the inner loop takes no more than 2 ms. It uses the *next* prefix to state that, starting at every visit to the marker start_inner, the next visit to end_inner should occur no later than 2 ms.

The second constraint:

$$\mathsf{time(outer)} - \mathsf{time(prev.outer)} \geq 10,$$

considers successive visits to the marker outer. The constraint requires that two successive visits to this marker be separated by at least 10 ms.

(a)	(b)
for $(i = 0;\ i < 10;\ i{+}{+})$ { **for** $(j = 0;\ j < 10;\ j{+}{+})$ /*! start_inner */ $a[i][j]$ += $i + j$ /*! end_inner */ ; *output*(*chan1*, $a[i][0]$); /*! outer */ }	**Requirements:** $\mathsf{time(next.end\_inner)} - \mathsf{time(start\_inner)} \leq 2$; $\mathsf{time(outer)} - \mathsf{time(prev.outer)} \geq 10$

Figure 5: Repeatedly visited markers.

We used the *next* prefix for the first constraint and the *prev* prefix for the second constraint, in order to illustrate the flexibility of the language. For this example, the two prefixes are completely interchangeable, and the same two constraints can be expressed using the alternative forms:

$$\begin{array}{lcll} \mathsf{time(end\_inner)} & - & \mathsf{time(prev.start\_inner)} & \leq \ 2 \\ \mathsf{time(next.outer)} & - & \mathsf{time(outer)} & \geq \ 10 \end{array}$$

2.7 The first/last visits to markers within a loop

It is frequently necessary to refer to the first or last visit to marker which resides inside a loop. In Figure 6, we illustrate how such references are managed, using the *next*/*prev* prefixes.

(a)	(b)
/*! initiate */ **while** ($e1$) { S1; /*! inner */ **while** (e2) { S2; S3; } S4; }; /*! terminate */	**Requirements:** $\mathsf{time(next.inner)} - \mathsf{time(initiate)} \leq 100$; $\mathsf{time(terminate)} - \mathsf{time(prev.inner)} \leq 200$

Figure 6: Example using first and last events.

The first constraint associated with this program bounds the time difference between the visit to initiate and the first visit to inner by 100. The second constraint bounds the time difference between the last visit to inner and the (only) visit to terminate by 200.

2.8 Side-Conditions in TimeC

(a)	(b)
for $(i = 0;\ i < 10;\ i{+}{+})$ { **if** $(i\%3 == 0)$ *output(chan1, i)* /*! out1(i) */; **if** $(i\%2 == 0)$ *output(chan1, i)* /*! out2(i) */; }	**Requirements:** $\mathsf{time}(\mathsf{next.out2}[i = m]) - \mathsf{time}(\mathsf{out1}[i = m]) \leq 2\,;$ $\mathsf{time}(\mathsf{next.out1}[i = n + 1]) - \mathsf{time}(\mathsf{out1}[i = n]) \leq 2$

Figure 7: Side-Condition constraints.

In TimeC, side-conditions may be used to restrict our attention to a subset of visits to a marker at which the program variables satisfy additional conditions. Any boolean expression allowed in C is admissible as a side-condition. A side condition may refer to the program variables as well as to logical variables which are used to relate the program variables at different visits. The example in Figure 7 illustrates the use of side-conditions. In this example, the specfications contains two contraints. The first constraint:

$$\mathsf{time}(\mathsf{next.out2}[i = m]) - \mathsf{time}(\mathsf{out1}[i = m]) \leq 2$$

states that there must be at most 2 time units of delay between the two output statements within each iteration of the loop. This constraint is active only when $i = 0$ or 6. Note that m here is a free logical variable and is implicitly universally quantified over each inequality.

The second constraint:

$$\mathsf{time}(\mathsf{next.out1}[i = n + 1]) - \mathsf{time}(\mathsf{out1}[i = n]) \leq 5$$

states that there must be at most 5 time units of delay between the first output statement in adjacent iterations. Note that this constraint is never active (always vacuous) since marker out1 is never visited in two consecutive iterations of the loop.

The side-conditions appearing in these constraints refer to the program variable i and to the logical variables m and n. As we see in the example, it is required that all program variables to which a side-condition refers appear as arguments to the annotation of the marker to which the side-condition applies. For the example of Figure 7, both out1(i) and out2(i) identify the program variable i as a variable that may appear in a side-condition that refers to them.

Side-Conditions may appear within brackets as shown in the example above. An alternative syntax allows a refence to a marker with expressions over the logical variables as arguments. For example, the two constraints specified above, can be equivalently written as

$$\begin{array}{lcll} \mathsf{time}(\mathsf{next.out2}(m)) & - & \mathsf{time}(\mathsf{out1}(m)) & \leq & 2 \\ \mathsf{time}(\mathsf{next.out1}(n+1)) & - & \mathsf{time}(\mathsf{out1}(n)) & \leq & 5. \end{array}$$

We refer to this presentation of side-conditions as *positionally conditioned* marker expressions. Thus, for a marker declared as $M(x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are program variables and $e_1, \ldots, e_k$ are expressions over logical variables, we interpret the marker expression $M(e_1, \ldots, e_k)$ as the following abbreviation:

$$M(e_1, \ldots, e_k) \quad = \quad M[x_1 = e_1 \ \&\& \ \cdots \ \&\& \ x_k = e_k]$$

2.9 First and last operators

In order to express certain "disjunctive" timing properties on programs, we introduce the operators first and last. The following example demonstrates the need for such operators:

```
for (i = 0; i < n; i++)
{  L1: { M1(i); S1;
         if (x1 == 1) { S2; M2(i); }
       }
   L2: { M3(i); S3;
         if (x2 == 2) { S4; M4(i); }
       }
}
```

Given the above program fragment, we want to state that in each iteration, the last event in the compound statement L1 should be at most t time steps away from the last event in L2. Events are uncontrained between iterations.

Using the last operator, we can state the above constraints as:

$$\mathsf{time}(\mathsf{last}(\mathsf{next.M3}(n),\ \mathsf{next.M4}(n))) \ - \ \mathsf{time}(\mathsf{last}(\mathsf{M1}(n),\ \mathsf{M2}(n)) \quad \leq \quad t$$

In a similar way, we can use the first operator to specify a requirement which constrains the time distance between the first event in the compound statement L1 and the first event in L2.

2.10 Syntax of TimeC constraints

The general syntax of TimeC constraints and assertions is presented in figure 8.

⟨*arg-list*⟩	::=	⟨*expression*⟩ | ⟨*expression*⟩, ⟨*arg-list*⟩
⟨*marker-ref*⟩	::=	⟨*marker-id*⟩ | ⟨*marker-id*⟩[⟨*side-condition*⟩] | ⟨*marker-ref*⟩(⟨*arg-list*⟩)
⟨*ref-list*⟩	::=	time(⟨*marker-ref*⟩) | time(⟨*marker-ref*⟩), ⟨*ref-list*⟩
⟨*first-last*⟩	::=	first | last
⟨*basic-ref*⟩	::=	⟨*marker-ref*⟩ | ⟨*first-last*⟩(⟨*ref-list*⟩)
⟨*present-ref*⟩	::=	time(⟨*basic-reference*⟩)
⟨*next-ref*⟩	::=	time(next.⟨*marker-ref*⟩)
⟨*prev-ref*⟩	::=	time(prev.⟨*marker-ref*⟩)
⟨*diff*⟩	::=	⟨*present-ref*⟩ − ⟨*present-ref*⟩ | ⟨*next-ref*⟩ − ⟨*present-ref*⟩ | ⟨*present-ref*⟩ − ⟨*prev-ref*⟩
⟨timing-constraint⟩	::=	⟨*diff*⟩ $\leq t$ | ⟨*diff*⟩ $\geq t$
⟨requirement-list⟩	::=	⟨timing-constraint⟩ | ⟨timing-constraint⟩ ; ⟨requirement-list⟩
⟨requirements⟩	::=	**Requirements:** ⟨requirement-list⟩

Figure 8: Syntax of TimeC constraints.

A *marker reference* is either a marker identifier or a marker identifier constrainted by a side-condition expression. A *basic reference* is a reference to a marker reference. A *relative reference* is a marker identifier prefixed with one of the prefixes *next* or *prev*. A *difference* is a difference between two basic references or between a basic reference and a relative reference. A *timing constraint* is an upper or lower bound on a difference. Timing constraints can be combined into a *requirement-list* which consists of one or more constraints separated by a semi-colon. Prefixed by the keyword **Requirements**, this forms the requirement part of a specification.

2.11 Program assumptions

In addition to timing assumptions, the user can also provide information on the bound on iteration counts using *program assumptions* (see figure 9). In the simpler cases, these can be expressed as bounds (lower or upper) on program-declared variables. A more involved case is when we wish to bound the number of iterations of a *while* loop, where there is no explicit program variable which is incremented on each iteration of the loop. To handle these cases, we add to the set of our annotations a *counter declaration*, which has the following syntax:

$\langle$counter-declaration$\rangle$ ::= /* counter: $\langle$identifier$\rangle$ */

A counter declaration should always prefix a *for* or a *while* statement. In Figure 9, we present the syntax of program assumptions, which, together with the timing assumptions complete the definition of a full *specification*.

$\langle$constrained-var$\rangle$	::=	$\langle$program-variable$\rangle$ | $\langle$counter$\rangle$	
$\langle$program-assumption$\rangle$	::=	$a \leq \langle$constrained-var$\rangle$ | $\langle$constrained-var$\rangle \leq b$	$a, b \in \mathbf{N}$
$\langle$assumption$\rangle$	::=	$\langle$timing-constraint$\rangle$ | $\langle$program-assumption$\rangle$	
$\langle$assumption-list$\rangle$	::=	$\langle$assumption$\rangle$ | $\langle$assumption$\rangle$; $\langle$assumption-list$\rangle$	
$\langle$assumptions$\rangle$	::=	**Assumptions:** $\langle$assumption-list$\rangle$	
$\langle$specification$\rangle$	::=	$\langle$requirements$\rangle$ | $\langle$assumptions$\rangle$ $\langle$requirements$\rangle$	

Figure 9: Program assumptions.

For example, if trials is a loop counter on a while loop, then asserting $5 \leq$ trials ≤ 10 provides information to the system that the loop always iterates between 5 to 10 times every time times is executed. This type of information is very useful for computing the upper- and lower-bounds on the timing intervals between two events, when the two events are separated by loops.

2.12 An example program

To demonstrate how all the features of TimeC presented in this section can be used together, let's return to the example schematically presented in figure 2(b) on page 4. We present the outline of a potential implementation for the Arrythmia monitor in C in figure 10(a). The program is annotated with markers start, start_gathering_data, end_gathering_data, alarm_start and alarm_end. Suppose the routine gather_data is a system call that performs I/O, and each call completes in 10 msec. Since this information is unknown to the scheduler, the user must provide this information to the system using assumptions[2] since otherwise it is impossible for the system to ensure that each iteration takes at most 30msec. The set of TimeC constraints and assumptions that implement this monitor loop is presented in figure 10(b).

3 Semantics of TimeC

Having defined the TimeC syntax of program annotations and specifications, we proceed to describe the semantics of an *annotated system*, examples of which are the annotated *C* programs presented in the previous

[2]It may be possible for the system to obtain this information through indirect means such as profiling.

```
void arrythmia_monitor()
{
    do {
        int data;
        int status;
        /*! start_gathering_data */
        data = gather_data();
        /*! end_gathering_data */
        code for analyzing data ...;
    } while (status == OK);

    /*! alarm_start */
    code for sounding alarm ...
    /*! alarm_end */
}
```

(a)

Assumptions:
time(end_gathering_data) - time(prev.start_gathering_data) ≤ 10

Requirements:
time(start_gathering_data) - time(prev.start_gathering_data) ≤ 30 ;
time(alarm_end) - time(alarm_start) ≤ 500

(b)

Figure 10: An example C program annotated with TimeC.

section. We model time as the natural numbers $\mathbf{N}$ (including 0), since the underlying machine hardware is synchronized with a global clock which is advanced by cycles .

In this section we introduce the basic constructs associated with an annotated system and show how they are used in order to describe the system's semantics. Using standard techniques, this description can be elaborated into a definition of the full formal semantics of annotated systems. We will illustrate these concepts on the sample program fragment of Figure 11. For convenience, we have numbered the lines in this program.

```
1.   /*! initiate */
2.   x = 1;
3.   for (i = 1; i <= 6; i++)
4.       {x = x * i;
5.        output(x)
6.        /*! inner */; };
7.   /*! terminate */
```

Figure 11: A sample program.

System Variables

We assume that the system is associated with a set of *system variables*. For the annotated-C programs of the previous section, these include all the program-declared variables plus the counters introduced by counter declarations. In order to define computations of such a program, we usually consider some additional variables which represent the control state of the program/system. In an annotated-C program, it is enough to have a single "program-counter" control variable π which points to the next statement (or marker annotation) to be executed.

For example, the sample program of Figure 11 has the system variables $\{x, i, \pi\}$, where x and i are program-declared variables and π is a control variable ranging over $\{1, \ldots, 7\}$. For example, the value $\pi = 4$ implies that the program is next ready to execute the statement $x = x * i$.

States and Situations

A system state is a *valuation* of all the system variables, including the counters and control variables, assigning to each variable a value in its domain[3]. For example,

$$\langle x{:}\, 6,\ i{:}\, 3,\ \pi{:}\, 5\rangle$$

represents a state of the sample program when it is ready to perform the *output* operation on the third iteration of the *for* loop.

We define a *situation* to be a time-stamped state. That is, a situation consists of a state and a natural number representing the value of the real-time clock when this state was observed. For example,

$$\langle x{:}\, 6,\ i{:}\, 3,\ \pi{:}\, 5,\ T{:}\, 17\rangle$$

represents the same state being observed at time $T = 17$.

For a situation s and a system variable v, we denote by $s[v]$ the value s assigns to v. Thus, if $s = \langle x{:}\, 6,\ i{:}\, 3,\ \pi{:}\, 5,\ T{:}\, 17\rangle$ then $s[x] = 3$ and $s[\pi] = 5$. We also write $s[T] = 17$ to denote that s observed at time 17.

Actions and Action Instances

Every annotated system identifies a set of *actions* which cause the system to change its state and perhaps to perform some communication with its environment. In an annotated-C program, every statement is an action. In general, every action may take a nonnegative amount of time to complete its execution. We consider marker annotations to be *virtual actions* in that they only change the value of the control variable and are assumed to take zero time. In a similar way, we consider every counter declaration to be another vitrtual action which takes zero time. We identify some actions to be *observable*. All marker and counter actions are observable. In addition, we usually include all input/output actions and all external-functions calls among the observable actions.

We define an *action instance* as follows:

1. For all actions (including virtual actions derived from markers) of the form $name(e_1, \ldots, e_k)$, we define an action instance to be an expression of the form

   $$name(e_1{:}\, v_1, \ldots, e_k{:}\, v_k),$$

   where $v_1, \ldots, v_k$ are values from the domains over which the expressions $e_1, \ldots, e_k$ range.

2. An action instance for the counter action associated with the declaration /* counter: x */ assumes the form

   $$count(x{:}\, a),$$

   where $a \in \mathbf{N}$.

3. For all other actions α, the action instance is identical to the action itself, i.e., α.

For simplicity, we assume that all marker and counter declarations appear on separate lines which do not contain any other statement.

[3]For simplicity, we will assume that there is no procedure call in the program. It is easy to extend the notion of control state model the stack and thus procedure call/return.

Computations

A *computation* of an annotated system is an interleaved sequence of situations and time-stamped action instancess, presented as

$$\sigma\colon \quad s_0 \xrightarrow{\alpha_1,t_1} s_1 \xrightarrow{\alpha_2,t_2} s_2 \xrightarrow{\alpha_3,t_3} \cdots,$$

where $s_0, s_1, \ldots$ are situations, $\alpha_1, \alpha_2, \ldots$ are action instances and $0 \leq t_1 \leq t_2 \leq \cdots$ are time instants at which the actions were initiated.

Such a sequence is defined to be a computation if it satisfies the following conditions:

- The first state s_0 is initial and has its program counter set to 1, implying that execution starts at line 1 of the program.

The remaining conditions consider the fragment $s_i \xrightarrow{\alpha_i,t_i} s_{i+1}$ of the computation, for every $i = 0, 1, \ldots,$ where we assume that $s_i[T] = t_i$, $s_{i+1}[T] = t_{i+1}$, and $s_i[\pi] = \ell$.

- For the case that $\alpha_i = \mathit{name}(e_1\colon v_1, \ldots, e_k\colon v_k)$, where *name* is the name of a function or a marker, we require that $v_1 = s_i[e_1], \ldots, v_k = s_i[e_k]$, and that s_i agrees with s_{i+1} on the interpretation of all system variables, except for π, which s_{i+1} interprets as $\ell + 1$. If *name* is a marker, we require that $t_{i+1} = t_i$. Otherwise, $t_{i+1} > t_i$.
- For the case that $\alpha_i = \mathit{count}(x\colon a)$, we distinguish between two cases according to the way the execution reached line ℓ on entering situation s_i.
  - If execution reached ℓ from a preceding line, then $a = 1$.
  - Otherwise (i.e. execution reached ℓ from a succeeding line implying repetition of the loop that x counts), we require that $a = s_i[x] + 1$

  In both cases, it is required that $t_{i+1} = t_i$ and that s_i agrees with s_{i+1} on the interpretation of all system variables, except for π and x. For these two system variables $s_{i+1}[\pi] = \ell + 1$ and $s_{i+1}[x] = a$.
- For all other actions, it is required that s_{i+1} be the situation derived from s_i by the execution of the action α_i and that $t_{i+1} > t_i$.

For example, the following is a possible computation of the sample program from Figure 11:

$$\begin{array}{cccccccc}
\langle x\colon -,\, i\colon -,\, \pi\colon 1,\, T\colon 0\rangle & \xrightarrow{\textsf{initiate},0} & \langle x\colon -,\, i\colon -,\, \pi\colon 2,\, T\colon 0\rangle & \xrightarrow{x=1,0} & \langle x\colon 1,\, i\colon -,\, \pi\colon 3,\, T\colon 1\rangle & \xrightarrow{\textbf{for}\ldots,1} \\
\langle x\colon 1,\, i\colon 1,\, \pi\colon 4,\, T\colon 2\rangle & \xrightarrow{x=x*i,2} & \langle x\colon 1,\, i\colon 1,\, \pi\colon 5,\, T\colon 3\rangle & \xrightarrow{\mathit{output}(x:1),3} & \langle x\colon 1,\, i\colon 1,\, \pi\colon 6,\, T\colon 4\rangle & \xrightarrow{\textsf{inner},4} \\
\langle x\colon 1,\, i\colon 1,\, \pi\colon 3,\, T\colon 4\rangle & \xrightarrow{\textbf{for}\ldots,4} & \langle x\colon 1,\, i\colon 2,\, \pi\colon 4,\, T\colon 5\rangle & \xrightarrow{x=x*i,5} & \langle x\colon 2,\, i\colon 2,\, \pi\colon 5,\, T\colon 6\rangle & \xrightarrow{\mathit{output}(x:2),6} \\
\langle x\colon 2,\, i\colon 2,\, \pi\colon 6,\, T\colon 7\rangle & \cdots & \langle x\colon 720,\, i\colon 6,\, \pi\colon 3,\, T\colon 19\rangle & \xrightarrow{\textbf{for}\ldots,19} & \langle x\colon 720,\, i\colon 7,\, \pi\colon 7,\, T\colon 20\rangle & \xrightarrow{\textsf{terminate},20} \\
\langle x\colon 720,\, i\colon 7,\, \pi\colon 8,\, T\colon 21\rangle & & & & &
\end{array}$$

Traces

Let P be an annotated program and let

$$\sigma\colon \quad s_0 \xrightarrow{\alpha_1,t_1} s_1 \xrightarrow{\alpha_2,t_2} s_2 \xrightarrow{\alpha_3,t_3} \cdots,$$

be a computation of P. The *timed trace* corresponding to σ is the sequence of time-stamped observable action instances. The *trace* corresponding to σ is the sequence of observable action instances, where we omit the time stamps.

For example, the timed trace corresponding to the previously displayed computation of the sample program (assuming that the observable actions are the markers and the *output* statements) is given by

$$\begin{array}{llllll} \langle \mathsf{initiate}, 0\rangle, & \langle \mathit{output}(x{:}\,1), 3\rangle, & \langle \mathsf{inner}, 4\rangle, & \langle \mathit{output}(x{:}\,2), 6\rangle, & \langle \mathsf{inner}, 7\rangle, & \ldots, \\ & \langle \mathit{output}(x{:}\,720), 18\rangle, & \langle \mathsf{inner}, 19\rangle, & \langle \mathsf{terminate}, 20\rangle. & & \end{array}$$

The (untimed) trace corresponding to the same computation is given by

$$\mathsf{initiate},\ \mathit{output}(x{:}\,1),\ \mathsf{inner},\ \mathit{output}(x{:}\,2),\ \mathsf{inner},\ \cdots,\ \mathit{output}(x{:}\,720),\ \mathsf{inner},\ \mathsf{terminate}.$$

We define the *timed semantics* of an annotated program P to be the set of all timed traces derived from computations of P. Similarly, we define the *(untimed) semantics* of P to be the set of all (untimed) traces derived from computations of P.

3.1 Satisfaction of a Specification by a Timed Trace

Let σ be a timed trace of the annotated program P. We proceed to define the conditions under which the timed trace σ satisfies a given specification.

As described in Section 2, the specification is partitioned into assumptions and requirements. Each requirement is a conjunction of timing-constraints. The assumptions are further partitioned into program assumptions and timing assumptions, which are again timing-constraints.

The computation σ satisfies the *program assumption* $x \leq a$, if all action instances of the forms $\mathit{name}(\ldots, x{:}\,v, \ldots)$ and $\mathit{count}(x{:}\,v)$ within σ satisfy $v \leq a$. A similar definition applies to program assumptions of the form $a \leq x$.

Thus, all computations of the sample program of Figure 11 satisfy the program assumption

$$1 \leq x \leq 1000.$$

For convenience, we treat all non-conditional marker references M as $M[\mathsf{true}]$, i.e., as conditional marker with the trivial condition true. Let L denote the set of all the logical variables appearing in the specification of the annotated program P. Let η be a valuation which assigns to each variable $y \in L$ a value in the domain of y. We say that the action instance $M_1(x_1{:}\,v_1, \ldots, x_k{:}\,v_k)$ *η-matches* the marker reference $M_2[\varphi]$ if $M_1 = M_2$ and φ evaluates to *true* over the combined interpretation which interprets each x_i as v_i and each $y \in L$ as $\eta[y]$.

We say that the action instance α_i *η-matches* the basic reference $\mathsf{first}(M_1[\varphi_1], \ldots, M_k[\varphi_k])$, if α_i η-matches one of the references $M_1[\varphi_1], \ldots, M_k[\varphi_k]$ and it is the action instance with the smallest index (i) which does so.

In a similar way, the action instance α_i *η-matches* the basic reference $\mathsf{last}(M_1[\varphi_1], \ldots, M_k[\varphi_k])$, if α_i η-matches one of the references $M_1[\varphi_1], \ldots, M_k[\varphi_k]$ and it is the action instance with the largest index (i) which does so, implying that σ contains only finitely many action instances η-matching one of these references.

Let α_i and α_j be two actions instances appearing in the timed trace σ, where $i < j$. We define as follows:

- The pair (α_i, α_j) *η-matches* the pair of basic references (b_1, b_2) if α_i η-matches b_1 and α_j η-matches b_2.
- The pair (α_i, α_j) *η-matches* the pair of references $(b_1, \mathsf{next}.M_2[\varphi_2])$, where b_1 is a basic reference, if α_i η-matches b_1, α_j η-matches $M_2[\varphi_2]$, and j is the smallest $k \geq i$ such that α_k η-matches $M_2[\varphi_2]$.
- The pair (α_i, α_j) *η-matches* the pair of references $(\mathsf{prev}.M_1[\varphi_1], b_2)$, where b_2 is a basic reference, if α_i η-matches $M_1[\varphi_1]$, α_j η-matches b_2, and j is the largest $k \leq i$ such that α_k η-matches $M_1[\varphi_1]$.

A timed trace σ *satisfies* a timing constraint $\mathsf{time}(r_2) - \mathsf{time}(r_1) \sim t$ ($\sim$ being $\leq$ or $\geq$) if $t_j - t_i \sim t$ for every two timed actions instances $\langle \alpha_i, t_i\rangle$ and $\langle \alpha_j, t_j\rangle$, $i < j$, appearing in σ, such that (α_i, α_j) η-matches the pair of references (r_1, r_2), for some L-interpretation η.

For example, the displayed computation of the sample program satisfies the following timing constraints:

$$\begin{array}{lcllclcllcl}
\mathsf{time(inner)} & - & \mathsf{time(initiate)} & \geq & 4 & \quad & \mathsf{time(terminate)} & - & \mathsf{time(inner)} & \leq & 16 \\
\mathsf{time(next.inner)} & - & \mathsf{time(inner)} & \leq & 3 & & \mathsf{time(next.inner)} & - & \mathsf{time(initiate)} & = & 4 \\
\mathsf{time(inner)} & - & \mathsf{time(prev.inner)} & \leq & 4 & & & & & & \\
\mathsf{time(terminate)} & - & \mathsf{time(prev.inner)} & = & 1 & & & & & &
\end{array}$$

Note that an equality can be derived as a conjunction of $\leq$ and $\geq$.

Given an annotated program P and a specification S consisting of an assumption φ and a requirement ψ, we say that P satisfies S (equivalently, that S is valid over P) if every timed trace of P which satisfies φ also satisfies ψ.

For example, the sample program of Figure 11 satisfies the specification given by

$$\begin{array}{lllr}
\text{Assumptions:} & \mathsf{time(next.inner)} - \mathsf{time(initiate)} & \leq & 4 \\
 & \mathsf{time(next.inner)} - \mathsf{time(inner)} & \leq & 3 \\
 & \mathsf{time(teminate)} - \mathsf{time(prev.inner)} & \leq & 1 \\
\text{Requirement:} & \mathsf{time(terminate)} - \mathsf{time(initiate)} & \leq & 20
\end{array}$$

Note that the timed trace

$$\langle \mathsf{initiate}, 0\rangle, \quad \langle output(x\colon 1), 3\rangle, \quad \langle \mathsf{inner}, 10\rangle, \quad \ldots, \quad \langle \mathsf{terminate}, 50\rangle$$

also satisfies the same specificaion, even though its execution time is 50, which violates the requirement $\mathsf{time(terminate)} - \mathsf{time(initiate)} \leq 20$. The reason that this is still satisfactory is that the same timed trace also violates one of the assumptions, namely, the assumption $\mathsf{time(next.inner)} - \mathsf{time(initiate)} \leq 4$.

3.2 Timely Implementations

A user of the methodology advocated in this paper will typically write an annotated-C program P and provide a corresponding TimeC specification S. The optimizing compiler described in this paper will read these inputs and produce an object code which is expected to run on a target RISC architecture and satisfy the specification. The combination of the produced object code and the platform on which it runs can be viewed as another annotated system T, with its own computations, traces and timed traces. There are two requirements we expect T to satisfy:

- T should correctly implement the source program P. That is, T should produce the same sequence of external communications and marker visits as P.
- T should satisfy the specification S.

To make these expectations more precise, we define an annotated system T to be an *implementation* of annotated system P if every (untimed) trace of T is also a trace of P.

Note that in comparing the source P and the target T, we completely ignore the questions of timing, and do not require that P satisfy the specification S. This is because P is envisaged as running on an abstract C machine about which we have no timing information. In fact the only way we can make reasonable timing claims about program P is by adding enough timing information to the *assumptions* part of the specification.

Given a pair (P, S), consisting of an annotated source program P and a specification S, we say that the annotated system T is a *timely implementation* of (P, S) if T is an implementation of P and T satisfies S.

For example, consider the program-specification pair, presented in Figure 12. Since the specification will be applied only to the target architecture, there is no need to give as assumptions any timing information related to the operations performed by the program itself. The only assumption needed is about the execution time of the external function *output*, which we stipulated to be bounded by 1.

(a)	(b)
/*! initiate */ $x = 1$; **for** ($i = 1$; $i <= 6$; i++) {$x = x * i$; /* begin_output */; $output(x)$ /* end_output */; }; /*! terminate */	**Assumptions:** time(end_output) − time(prev.begin_output) ≤ 1 **Requirements:** time(teminrate) − time(initiate) ≤ 20

Figure 12: Sample program with Specification.

4 Tractable subset of TimeC: Time_tract

It is trivial to phrase undecidable problems in the current form of TimeC. Thus for practical use, it is often desirable to restrict TimeC to a "tractable" subset so that implementations of a program can be effectively derived from the timing specifications. We call such a subset Time_tract.

Intuitively, Time_tract constraints are restricted to conditional upper- and lower-bounds between two events that are related by a fixed iteration difference in the source program. To realize Time_tract, we impose the following syntactic and semantic restrictions on TimeC constraints:

1. Constraints of the form

   $$\mathsf{time}(\mathsf{M}[P]) \sim t$$

   and

   $$\mathsf{time}(\mathsf{M}_1[P_1]) - \mathsf{time}(\mathsf{M}_2[P_2]) \sim t$$

   are disallowed in Time_tract. Only the remaining two forms of relative constraints

   $$\mathsf{time}(\mathsf{next.M}_1[P_1]) - \mathsf{time}(\mathsf{M}_2[P_2]) \sim t$$

   and

   $$\mathsf{time}(\mathsf{M}_1[P_1]) - \mathsf{time}(\mathsf{prev.M}_2[P_2]) \sim t$$

   are allowed.

2. Predicates in Time_tract are restricted to conjunctions of simple equality tests of the form $u = i + c$, where i is a program loop index variable on *well-formed* loops, c is some integer constant, and u a free variable.

   A well-formed loop has the following properties: (i) it is either a loop in the outermost level or nested perfectly within another well-form loop; and (ii) it must be a simple loop with a single index i such that i increments by a constant amount in every iteration.

3. Given a marker M, reference to an event associated with M must be uniquely identified by the indices of the loops in which M is nested under. Furthermore, relative constraints of the form

   $$\mathsf{time}(\mathsf{next.M}_1[P_1]) - \mathsf{time}(\mathsf{M}_2[P_2]) \sim t$$

for $(i = 0;\ i < 10;\ i{+}{+})$ { **for** $(j = 0;\ j < i;\ j{+}{+})$ { **if** $(j\%3 == 0)$ *output*(*chan1*, i) /*! out1(i,j) */; **if** $(j\%2 == 0)$ *output*(*chan1*, i) /*! out2(i,j) */; } } (a)	**illegal** Time_tract **constraints:** $\mathsf{time}(\mathsf{next.out2}[m = j]) - \mathsf{time}(\mathsf{out2}[m = j]) \leq 2$ $\mathsf{time}(\mathsf{next.out1}[n = i + 1 \wedge m = j]) - \mathsf{time}(\mathsf{out1}[n = i \wedge m = j]) \leq 2$ **legal** Time_tract **constraints:** $\mathsf{time}(\mathsf{next.out2}[m = i \wedge n = j]) - \mathsf{time}(\mathsf{out2}[m = i \wedge n = j]) \leq 2$ $\mathsf{time}(\mathsf{next.out1}[m = i \wedge n = j + 1]) - \mathsf{time}(\mathsf{out1}[m = i \wedge n = j]) \leq 2$ (b)

Figure 13: Time_tract constraints

and

$$\mathsf{time}(\mathsf{M}_1[P_1]) - \mathsf{time}(\mathsf{prev.M}_2[P_2]) \sim t$$

can only be used to constraint the time delay between two events x and y only if the iteration difference separating x and y is a constant.

We illustrate these restrictions of Time_tract in Figure 13. The first constraint

$$\mathsf{time}(\mathsf{next.out2}[m = j]) - \mathsf{time}(\mathsf{out2}[m = j]) \leq 2$$

is ambiguous since the event reference out2$[m = j]$ may refer to visits of marker out2 when the loop index $i = 0, 1, 2, \ldots$ etc. On the other hand, constraint

$$\mathsf{time}(\mathsf{next.out1}[n = i + 1 \wedge m = j]) - \mathsf{time}(\mathsf{out1}[n = i \wedge m = j]) \leq 2$$

is unambiguous, but the iteration difference between the event associated with the first occurrence of out1 and the second is not fixed. In contrast, both constraints

$$\mathsf{time}(\mathsf{next.out2}[m = i \wedge n = j]) - \mathsf{time}(\mathsf{out2}[m = i \wedge n = j]) \leq 2$$

and

$$\mathsf{time}(\mathsf{next.out1}[m = i \wedge n = j + 1]) - \mathsf{time}(\mathsf{out1}[m = i \wedge n = j]) \leq 2$$

are valid Time_tract constraints.

These restrictions make the timing constraint verification problem much easier to solve. We will discuss these issues further in section 5.1.2 and section 6.2.

5 Instruction scheduling of TimeC programs

We now refocus our attention on instruction scheduling of TimeC programs. Our basic framework of *program synthesis* is presented in figure 14. Here, a high level program annotated with TimeC constraints is first transformed into a low level intermediate representation (IR); at the same time this process also responsible for translating TimeC constraints imposed on the high level language constructs into constraints imposed on

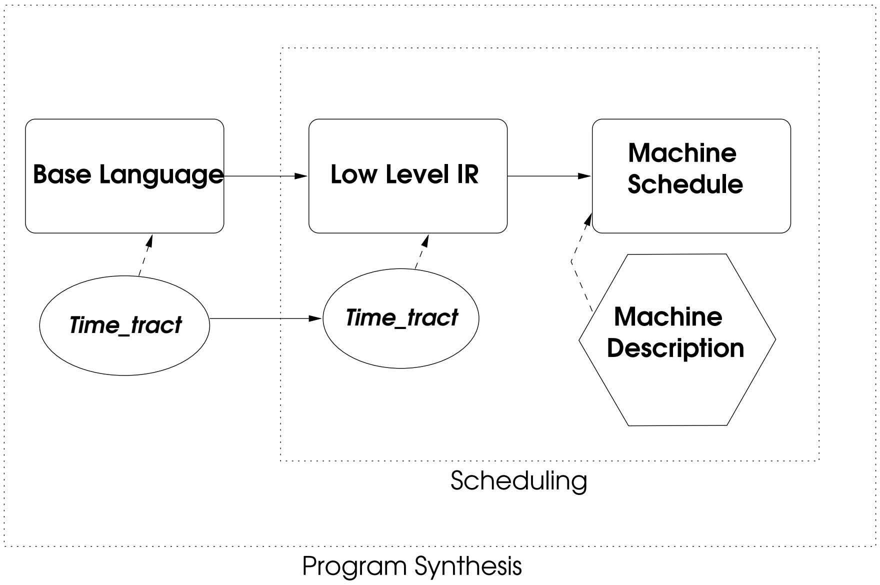


Figure 14: Framework of TimeC program synthesis.

the low level IR. Due to this need, program transformations that drastically alter the structure of the program, such as loop transformations, are not used.

Instruction scheduling refers to the process of generating a feasible machine schedule from the lower level IR. Timing information necessary for this process is derived from two sources (i) the machine model of the target processor, and (ii) timing assertions provided by the application programmers. In general, this process may involve reordering and replicating event-less instructions through out the program in order to satisfy the timing constraints.

For example, suppose we want to implement the data gathering routine in the Arrthymia monitor application presented in figure 2(b). Suppose also the monitoring sensor contains 3 I/O ports: the output port is for sending commands to the device and two input ports for receiving data. In order to get sensible data from the device, the following protocol is used: first, we must send a command **read** to the command channel, then read from both input channels within 4 msecs, but no sooner than .5msec after the command **read** is sent. Finally, we must send reset commands to the sensor before the next sampling. We know that all I/O commands complete in exactly 1 msec. In order to return useful result to the main application, the input data must also be rescaled and combined into meaningful units. These are performed using the routine *rescale* and *compute_result* respectively, which take at most 1msec each. Finally, since the main application requires the routine *gather_data* to complete in 10msec, we need to impose an extra constraint to state this fact.

In the first two columns of figure 15 we present the code for this data gathering routine and its associated constraints and assertions. It is clear that if the order of execution in the original program is followed exactly the timing constraints cannot be satisfied. Thus in order to achieve feasibility we may try to move the computation operations outside of the critical path, which in this case is determined by the time between the first *output* to the last *input* operation. The last column of figure 15 shows such a feasible schedule after scheduling.

As is standard in conventional global scheduling ([2], [9], [11], [28]), we treat loop scheduling [28], [22] and acyclic scheduling [2], [9], [11] separately. This division of tasks is traditionally used since combining both in the same scheduling framework leads to high degree of complexity in engineering and analysis. Furthermore, the two approaches are traditionally analyzed separately, and efficient and effective heuristics have been discovered that work on each domain individually. For the same reasons we also advocate taking this approach for scheduling programs instrumented with TimeC constraints.

```
int gather_data()
{
    int x, y, a, b;
    /*! +start */
    output(command,read);
    /*! +read1_start) */
    input(sensor1,&x);
    /*! -read1_finish */
    output(command,reset1);
    /*! +rescale_start */
    a = rescale(x);
    /*! -rescale_finish */
    /*! +read2_start) */
    input(sensor2,&y);
    /*! -read2_finish */
    output(command,reset2);
    /*! +compute_start */
    result = compute(a, y);
    /*! -compute_finish */
    return result;
    /*! -finish */
}
```

(original)

```
constraints:
    time(read1_finish) − time(start) ≤ 4;
    time(read2_finish) − time(start) ≤ 4;
    time(read1_start) − time(start) ≥ .5;
    time(read2_start) − time(start) ≥ .5;
    time(finish) − time(start) ≤ 10;
assertions:
    time(compute_finish) − time(compute_start) ≤ 1;
    time(rescale_finish) − time(rescale_start) ≤ 1;
```

(TimeC constraints)

```
int gather_data()
{
    output(command,read);
    input(sensor1,&x);
    output(command,reset1);
    input(sensor2,&y);
    output(command,reset2);
    a = rescale(x);
    result = compute(a, y);
    return result;
}
```

(a feasible schedule)

Figure 15: Scheduling to ensure timing feasibility.

5.1 Global scheduling preliminaries

5.1.1 Acyclic scheduling

The input to an acyclic global scheduler is very similar to the that for basic-blocks with a few essential extensions. First, because of branching, we must also be sensitive to the program's control-flow in addition to data-dependence. In particular, moving instructions out of their initial basic-blocks past program *branch* and *merge* points [1], [27] in the CFG lead to "side-effects" which require special attention and add to the complexities.

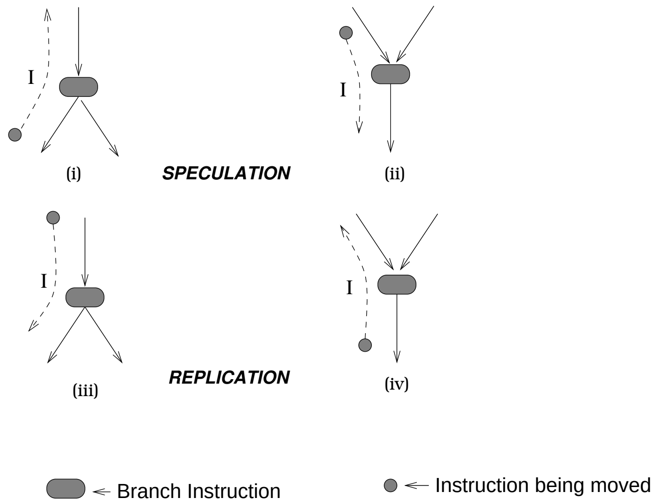


Figure 16: Speculation and replication.

A significant issue that arises in the context of moving instructions globally is the effects of *speculation* and *replication*. In either case these movements may require expensive repair, leading to *overheads*, as detailed in figure 16.

In cases (i) and (ii) of this figure, instruction I is moved outside of the branch/merge instruction and goes from being executed conditionally to begin execution unconditionally, even when its result is not needed. This type of an execution is referred to as *speculation*. The complementary situation is shown in the two figures in cases (iii) and (iv): after code motion, instruction I is executed only when the left branch is taken, whereas previously, it is executed independent of the branch. To preserve the semantics of the program, a duplicate of I is placed on the right branch as well — hence, the term *replication*. Both of these side-effects can lead to extra costs and consequently most compilers limit the degree to which this type of instructions can cross basic-block boundaries.

Global acyclic scheduling may also involve instruction movements that violate data dependences. Such instruction movements may be rendered legal by introducing new registers and inserting compensating copy instructions. For example, consider the control flow graphs in figure 17. The leftmost graph displays the initial untransformed program, while the middle graph displays the program after one of the statements of the right has been speculatively moved[4]. The rightmost graph displays the program after speculation has been applied to both branches. Notice that in order to render the movement legal, the destination register x from one of the instructions has been *renamed* to $x1$, and a *compensation copy* has been left behind in the left branch. This technique is called *register renaming* and in general can remove anti- and output- dependences from a program to allow a wider range of global instruction movements.

In our global scheduling model, we will permit instructions that are not *branches* or *merges* to be moved without any restrictions, provided that they are *safe*, i.e. they do not cause any traps or exceptions during

[4] We will assume that the instruction does not cause any runtime traps and exceptions.

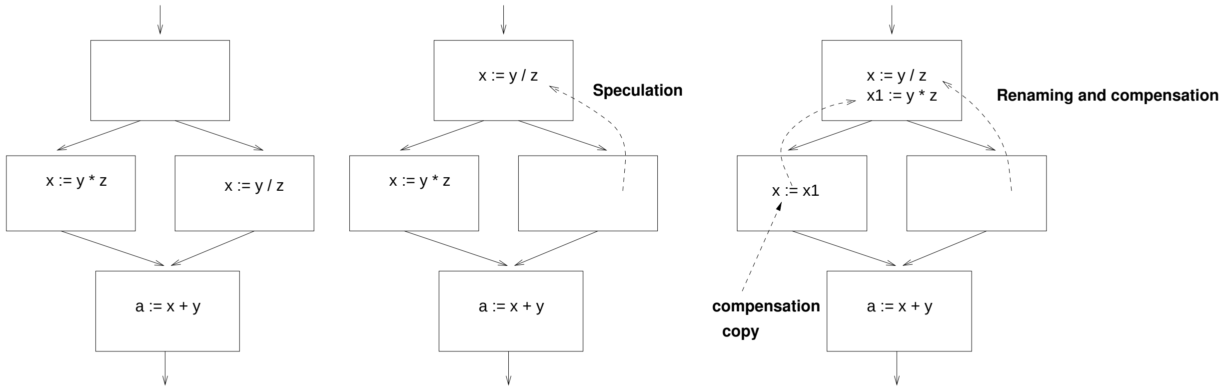


Figure 17: Speculation and register renaming.

execution. In contrast, branches (and merges) are treated differently since the corresponding overheads can be prohibitive [12]. Consequently, scheduling technology frequently limits the relative reordering of these instructions completely. We will retain this restriction in our subsequent discussion.

5.1.2 Feasible global acyclic schedule

Let us now make precise the notion of a *global acyclic schedule*. Consider an acyclic (possible branching) program region P with an instruction set $\mathbf{I}$. Such a program can be represented as an acyclic control flow graph (CFG) in which each node is a basic block, i.e. a subset of instructions from $\mathbf{I}$, including a branch instruction. Basic blocks with one successor are the merge nodes while basic blocks with two or more successors are the branching nodes. We will require the program P to have a single entry node; however, multiple sink nodes are allowed, each representing a different *exit point* of the program region. Each instruction i reads its input from some registers $r_1, r_2, \ldots$ and writes its output to registers $r'_1, r'_2, \ldots$ etc. Given a path through P, we define the *read set* as the set of registers that are read (before overwritten by some other instructions) by some instructions and the *write set* is the set of registers that are last-written by some instruction on the path.

In addition to the control flow graph, a precedence order $\sqsubset$ represented as the data dependence graph (DDG) is imposed on the instructions, and for each pair of instructions i and j such that $i \sqsubset j$, the *latency* function $w(i, j)$ defines the minimal number of machine cycles that must be present between the completion time of i and the start-time of j.

A *global acyclic schedule* for a program P on some machine M[5] is another DAG P' such that each node represents a basic block schedule, i.e. a mapping from instruction to time (relative to the start time of the basic block), and each edge represents the branching relations between basic blocks. Since replication are allowed, each instruction in P can be mapped into one or more instructions in P'. We call these *instances* of i, and denote each instance in the schedule P' with a unique subscript k, as in i_k. Notice that due to global instruction movement, the instances of i in the schedule P' may in fact be positioned in different basic blocks. Furthermore, since speculation may require register renaming, the sources and destination registers of an instruction i may be altered in P'. Finally, we allow any number of register-to-register copy instructions to be inserted in the schedule P' for speculative compensation.

We say that P' is a *feasible global acyclic schedule* for a program P if the following conditions are satisfied:

1. The precedence constraints are satisfied, i.e. for each pair of instructions i and j in P such that $i \sqsubset j$, there must be at least $w(i, j)$ time steps between any instances of i and j in P', whenever such pairs lie on the same execution path. We assume that external observable instructions are initially annotated with precedence constraints and preserving these precedence constraints implies that no observable events can be reordered during execution.

[5]We will assume the number of registers on the machine is unbounded during scheduling.

2. Each instance of the instruction i in the schedule P' must obtain its operands $o_1, o_2, \ldots, o_n$ from the same set of instructions j's as in the original program P. Notice that since we are allowed to rename registers, an operand of i in the schedule P' may be bound to a different intermediate register as in P. Furthermore, since register-to-register copy instructions may be introduced during scheduling, i may read from j indirectly through a sequence of these copies.

3. The resource constraints are satisfied, i.e. for each time step during the execution of the schedule, no more than m_t instructions are issued, where m_t refers to the number of functional units of type t in the machine M.

4. The branching instructions are in the same relative order in both P and P', i.e. branches are not reordered.

5. Given any execution path in P, instructions that appear in that path must also appear in the same path in P'.

6. Let M_i and M_j be markers associated with instructions i and j respectively. Whenever a TimeC constraints such as $\mathsf{time}(\mathsf{next.M}_i) - \mathsf{time}(\mathsf{M}_j) \sim k_{ij}$ appear (where $\sim$ is one of $\leq$ or $\geq$), and some instances of instructions i and j, lie on the same path in P', then it must be true that $\tau_i - \tau_j \sim k_{ij}$, where τ_i and τ_j are the execution times of the instances i and j in the schedule.

Together these conditions imply that the scheduled program P' preserves the functional semantics of program P and satisfies the timing-constraints imposed by TimeC.

5.1.3 Instruction scheduling for loops

Software pipelining [28], [22] is a common technique for exploiting instruction-level parallelism in loops by overlapping instructions multiple iterations together to hide the long latencies within an iteration. Unlike techniques such as loop unrolling, which tend to leave idle cycles in the end of loop iteration and thus may proceed in burst-like manner, software pipelining aims to achieve a steady and thus faster iteration rate by taking into account of the interaction between loop iterations. Typically, a software pipelined loop is decomposed into three parts, (i) a *prologue* which initializes the loops before it enters the steady state, (ii) the *kernel* which corresponds to the steady state of the loop, and (iii) the *epilogue* which completes the execution of a loop. In more complex schemes, it is possible to eliminate the need of an epilogue by speculatively executing the epilogue within the kernel. Regardless of the various methods of deal with prologs and epilogues, the main objective of software pipelining is to minimize the *initiation interval*, i.e. the length of the kernel.

for $(i = 0;\ i < n;\ i{+}{+})$
$X[i] = a{*}X[i];$

(a)

LD	X[r_i], r_1
MULT	r_a, r_1, r_2
ST	r_2, X[r_i]
ADD	r_i, 1, r_i

(b)

Cycle	**Operations**			
0	LD 0, r_{i1}	LD 1, r_{i2}	LD 2, r_{i3}	LD 3, r_{i4}
1	LD_1	nop	nop	nop
2	MULT_1	LD_2	nop	nop
3	ST_1	MULT_2	LD_3	nop
$4 \leq i \leq n$	ADD_{i-3}	ST_{i-2}	MULT_{i-1}	LD_i
$n+1$	nop	ADD_{n-2}	ST_{n-1}	MULT_n
$n+2$	nop	nop	ADD_{n-2}	ST_n
$n+3$	nop	nop	nop	ADD_n

(c)

Figure 18: . Software pipelining example. (a) C code. (b) Assembly in loop body. (c) Software pipelined loop.

As a simple example, consider the C loop presented in figure 18(a) and its associated assembly code in figure 18(b). Due the data dependences within a single iteration of a loop, it is not possible to speed up the loop without overlapping the executions of different iterations. For simplicity, we will assume in this example that the CPU can issue two load/store operations and two scalar operations in each cycle, and all operations completes in one cycle. Figure 18(c) shows a possible pipelined execution in which 4 adjacent iterations of the loop are overlapped. Cycles 0 to 3 constitute the prologue of the loop[6]; cycles 4 to n is the kernel of the main loop and its initiation interval is one cycle. Finally cycles $n+1$ to $n+3$ constitute the epilogue of the loop and it completes the executions for iterations $n-2$ to n.

The details of software pipelining and related techniques are outside the scope of this paper. However, observe that the main techniques for decomposing a complex program into manageable units are still applicable in the presence of TimeC constraints:

1. To deal with nested loops, we schedule loops in the order of nesting. Scheduled loops are then reduced to a single atomic operator in subsequent acyclic scheduling phases.
2. To deal with loops with internal branching, we can utilize techniques such as (i) *hierarchical reduction*[22], which collapses conditional constructs into atomic operators in an innermost to outermost order; and (ii) *if-conversion*[18]/*reverse-if-conversion*[33], which transform control dependences into data dependences and vice-versa.

6 Complexity of TimeC and Time_tract Constraints

6.1 Complexity of TimeC

TimeC constraints are undecidable since the halting problem can be trivially encoded as TimeC constraints. For example, consider the following annotated program fragment

$$S_1; \text{/*! } \mathsf{M}_1 \text{ */ } S_2 \text{ /*! } \mathsf{M}_2 \text{ */}$$

in which the externally observable statement S_2 is known to take 2 time units to execute. Then $\mathsf{time}(\mathsf{next.M}_2) - \mathsf{time}(\mathsf{M}_1) \leq 1$ is valid iff S_1 does not terminate.

Since even the simplest constraints are undecidable under the most general model, to get more informative results, we will assume in the remainder of this paper that all control flow paths in a program are accessible.

```
a = P(x);
for (i = 0; i < 1; i++) {
    if (i == 0)
        /*! M1 */ S1;
    if (a == i)
        /*! M2 */ S2;
    S3;
}
```

Figure 19: Unbounded TimeC complexity.

Under this restricted model, the complexity of TimeC is still unbounded. For example, consider the following program fragment in Figure 19. in which $P(x)$ is some arbitrary complicated predicate on x, and the externally observable statements S_1, S_2, and S_3 each take one time unit to execute[7]. Then the TimeC constraint

$$\mathsf{time}(\mathsf{next.M}_2) - \mathsf{time}(\mathsf{M}_1) \leq 2$$

[6]Notice that the loop induction register r_i has been split into 4 registers in the pipelined loop to eliminate its loop carried dependences.

[7]For simplicity we will assume the condition tests take 0 time.

is valid iff the predicate $P(x)$ returns 0, and TimeC constraint checking in this case is equal to complexity as computing $P(x)$.

6.2 Complexity of Time_tract

We now attempt to determine the complexity of Time_tract program synthesis. We assume that the transformation from a high level program language to the low level IR can be performed in polynomial time. With this assumption in place, we can state the Time_tract program synthesis problem in the following decision form:

Definition 1 (Program synthesis) *Given the specification of a program P in CFG and DDG form, timing constraints expressed in* Time_tract*, and a positive integer N. Does there exists a schedule P' that* (i) *satisfies the* Time_tract *constraints, and* (ii) *preserves the functional semantics of P, such that P' is of size $O(\min(N, n)^k)$, where n is the size of the input instance P and k is some predetermined constant?*

We will consider the two subproblems *(i)* and *(ii)* separately.

Subproblem *(i)* can be handled as follows. Informally, if the scheduled program is polynomially size bound then the number of timing constraints imposed on the scheduled program is also bound in the same manner. As before, we assume that all possible paths in the control flow graph are accessible. Since the input program is assumed to be structured, we can process loops from the innermost to the outermost. For each loop, we compute the minimal and maximal time interval needed to execute each iteration. Since the iteration difference between any two markers is a constant in Time_tract, it is trivial to use this information to determine the upper- and lower-bounds on the time interval between any two events in polynomial time. This in turn can be used to compute the satisfiability of the timing constraints in Time_tract. Since this process is simple in concept, we will omitted the details.

As to subproblem *(ii)*, its complexity depends entirely on the notion of "semantics equivalence" that we choose to adopt. In the general case, the program synthesis problem is clearly undecidable. Provided that we restrict the problem statement in some reasonable manner, however, NP-completeness of the program synthesis problem can be shown.

For example, we can restrict semantics equivalence to the following problem: given a program P and a schedule P', deciding whether P' is a feasible global schedule of P according to the restricted scheduling criteria specified in section 5.1.2. Recall that we require that no branches are reordered in the schedule P' in this definition. With this restriction, we can verify that each of the conditions stated in section 5.1.2 can be checked in polynomial time, and the program synthesis decision problem using this notion of "semantics equivalence" is NP-complete. We can also extend this complexity argument to the cyclic scheduling case with the same result, using essentially the same form of argument.

7 Comparison with other work

When comparing our work to previously proposed approaches for the specification of real-time constraints, we should distinguish between to classes of proposals: specification for analysis only, and specification leading to synthesis (scheduling).

Proposals belonging to the first class concentrate on the real-time specification of the system where the main goal is the development of a maximally expressive and natural language for the formulation of real-time requirements at the specification level. The only algorithmic tractability constraints considered by these proposals is the ability to check whether a given implementation satisfies the specified real-time requirements, i.e., the ability to analyze (verify) an existing system.

The second class of proposals, which includes the work reported here, does not stop at specification and analysis (verification), but proceeds to synthesize an implementation that satisfies the real-time requirements, if such an implementation exists.

Since synthesis is always more difficult than analysis, we should not be surprised that the languages proposed for the first class (specification and analysis only) are typically more powerful and expressive than the languages developed for the more ambitious class which also aims to synthesize satisfactory implementations.

7.1 TCEL

A notation for specifying time called TCEL and a framework for scheduling programs derived from TCEL specifications were introduced in [15]. Contrasting with our approach and results here, TCEL is a programming language wherein timing constraints are explicitly specified as part of the control-flow of the application program; in this sense, the programmer must write TCEL programs. The type of constraints allowed in TCEL are listed in figure 20.

$$\textbf{do } S_1 \text{ [}\textbf{start after } t_{\min}\text{] [}\textbf{start before } t_{max1}\text{] [}\textbf{finish within } t_{max2}\text{] } S_2$$

$$\textbf{every } p \textbf{ while } E \text{ [}\textbf{start after } t_{min}\text{] [}\textbf{start before } t_{max1}\text{] [}\textbf{finish within } t_{max2}\text{] } S$$

Figure 20: TCEL timing constraints.

Semantically, the "**do**" construct has the same effect as if S_1 then S_2 are executed sequentially, with the additional effect of imposing the following constraints:

- **start after** t_{min}: there is a minimal delay of t_{min} between the last event in S_1 and the first event in S_2.
- **start before** t_{max1}: there is a maximal delay of t_{max1} between the last event in S_1 and the first event in S_2.
- **finish within** t_{max2}: there is a maximal delay of t_{max2} between the last event in S_1 and the last event in S_2.

If the event in question is missing (for example, if during the execution of S_1 no observable events is encountered) then the constraints hold trivially.

Similarly, the "**every**" construct has the same functional semantics as the while loop. In addition, it executes each iteration of the loop in exactly p units of time, and imposes the following timing constraints on each iteration:

- **start after** t_{min}: there is a minimal delay of t_{min} between the start time of current iteration and the first event in S.
- **start before** t_{max1}: there is a maximal delay of t_{max1} between the start time of current iteration and the first event in S.
- **start before** t_{max1}: there is a maximal delay of t_{max1} between the start time of current iteration and the first event in S.
- **finish within** t_{max2}: there is a maximal delay of t_{max2} between the start time of current iteration and and the last event in S.

From this semantics description it is clear that TCEL is a strict subset of TimeC in terms of expressive power: first and last event of a region can be expressed using the min, max, first and last operators of TimeC. For example, the timing constraints in the TCEL statement

$$\textbf{do } S_1 \textbf{ finish within } t_{max2} \; S_2$$

can be expressed in TimeC as

$$\max_{e \in \mathsf{events}\ (S_1)} \mathsf{time}(e) - \max_{e' \in \mathsf{events}\ (S_2)} \mathsf{time}(e') \leq t_{max2}$$

where events (S) denotes the set of event expressions occurring in the statement S.

On the other hand, while TimeC allows the user to specify constraints that must hold across iterations of a loop, there is no equivalent construct in TCEL. Complexity-wise, using the same acyclic global semantics, program synthesis for acyclic TCEL programs can also be shown to be NP-complete.

7.2 RTL

Mok's Real Time Logic (RTL)[16, 17] is a system of program annotations for specifying constraints for task based languages such as Ada. Events in RTL are partitioned into four classes: *(i)* external event (written as ΩA where A is an action[8]), *(ii)* start event (written as $\uparrow A$) *(iii)* stop event (written as $\downarrow A$), *(iv)* and transition event. Start event and stop event are analogous to TimeC event markers associated to the beginning or end of a statement. Like TimeC, the domain of time is modeled by the natural numbers. All timing constraints are expressed in terms of a first order logic with equality extended with integer constants, $+, -, \leq$[9], and the built-in "occurrence function," written as @. The *partial* function $@ : \mathit{Event} \times \mathbf{N} \rightharpoonup \mathbf{N}$ is given the following interpretation:

$@(e, i) =$ time of the ith occurrence of event e. If no such occurrence exists, then $@(e, i)$ is undefined.

For example, to express the condition that the ith occurrence of event e_2 must start no less than k time units after the ith occurrence e_1 finishes (but event e_1 may be sporadic), we can write

$$\forall i. \forall t. @(\downarrow e_1, i) = t \Rightarrow t + k \geq @(\uparrow e_2, i)$$

Since quantification is over the natural numbers, $@(\uparrow e_2, i)$ must be defined whenever $@(\downarrow e_1, i)$ is defined.

The timing constraint validity checking procedure is to determine, given a set of implementation characteristics I and a set of correctness criteria S, both specified as conjunctions of RTL formulas, whether $I \Rightarrow S$ is valid. Since unrestricted RTL formulas involve first order arithmetic (without multiplication) and uninterpreted function symbols, it is in general undecidable. When restricted to "safety" assertions, an exponential time decision procedure for RTL formulas checking is known.

The expressive power of RTL is a strict superset of TimeC: addition with constants, min, max and first can be easily simulated by rewriting them into the corresponding RTL constructs. Counter variables and last can also be easily translated given the full power of first order logic. In the other direction, while the semantics of @ in RTL and time in TimeC are similar, RTL also allows disjunction, negation and existential quantification, which are missing in TimeC. Furthermore, in TimeC it is impossible to express interesting properties such as time-bounded response property and periodicity. For example, the constraint

$$\mathsf{time}(q) - \mathsf{time}(p) \leq u$$

does not guarantee that q will follow p in every instance of p— the event associated with p and/or q may be missing and the constraint is still deemed valid in our interpretation. In fact, an empty trace (corresponding to a program that never produce any events) trivially satisfies every TimeC constraint set. In contrast, RTL can express the time-bounded response property of events p and q as

$$\forall i. \forall t. @(p, i) = t \Rightarrow t + c \geq @(q, i)$$

(event q must follow event p within c time units), the periodicity of event p as

$$\forall i. \exists t. @(p, i) = t$$

[8] Actions are analogous to markers in TimeC.

[9] RTL also allows syntactic sugar such as multiplication by constants, but these are redundant in terms of expressibility.

(all occurrence of event p exists), and the fixed periodicity of event p as

$$\exists u.\forall i.\forall t.@(p,i) = t \Rightarrow @(p,i+1) = t+u$$

(p is periodic and each period is u units long.) None of these properties can be expressed as TimeC constraints.

While these real-time properties are important for the specification of general real-time systems, we believe that they are less important in our program synthesis framework.

7.3 CRL

Language for Complex Real-Time Systems (CRL)[31] is a real-time programming language with high level built-in features such as object-orientation, threading and exceptions. CRL uses CaRT-Spec[30] as an underlying specification language. System specifications are first presented in CaRT-Spec; CRL program stubs are generated from the specification by a translator tool. CRL programs are mapped into C++ through a program transformation/optimization process.

Timing constraints CRL are specified in terms of *labels* annotated within CRL programs. Given a statement S, the syntax $\$l_1!\ S\ !l_2\$$ allows the programmer to associate the labels l_1 and l_2 to the start and completion of statement S. Such an annotated statement is called timed in CRL. In this regard labels in CRL are essentially markers in TimeC.

Timing constraints in CRL, like TimeC, are divided into two classes: absolute, which relate a timed statement to the beginning or end of some frame of reference; and relative, which relate the interval between two timed statements. Timing constraints are attached to program statements using the **timeconstraint** construct (see figure 21). CRL constraints imposed on conditionally executed statements are trivially satisfied if the statements are not reached, as in TimeC's interpretation.

$$\begin{array}{lcl}
\textit{statement} & ::= & \textit{statement time_constraint} \\
\textit{time_constraint} & ::= & \textbf{timeconstraint} \\
 & & \left(\begin{array}{ll} & \textbf{use}\ \textit{absolute_time_expr} \\ \mid & \textbf{userelative}\ \textit{relative_time_expr} \\ \mid & \textbf{nosoonerthan}\ \textit{absolute_time_expr} \\ \mid & \textbf{nolaterthan}\ \textit{absolute_time_expr} \\ \mid & \textbf{nosoonerthanrelative}\ \textit{relative_time_expr} \\ \mid & \textbf{nolaterthanrelative}\ \textit{relative_time_expr} \end{array}\right)^{*} \\
 & & \textbf{endtimeconstraint} \\
 & & \\
\textit{absolute_time_expr} & ::= & \textit{integer} \mid (\textit{integer, label}) \\
\textit{relative_time_expr} & ::= & (\textit{label integer flags}) \mid (\textit{label integer flags, label})
\end{array}$$

Figure 21: CRL timing constraints.

In addition to timing constraints, a novel feature of CRL is that it allows the user to annotate the program with three types of assertions: (i) iteration assertions, (ii) recursion assertions and (iii) length assertions. Iteration assertions declare the upper and/or lower bounds on the number of iterations of some loop; recursion assertions declare the recursion depth of function calls; and length assertions declare the size limit of data contained in variables. All these assertions are used by the program transformation process to extract additional information to help satisfy the timing constraints.

7.4 Extending Hoare logic with time

The work of Shaw[29] is concerned with reasoning timing constraints and execution times of high level programming languages. This work takes the approach of extending Hoare logic[14] with the ability to reason with time by introducing global clock variables that are updated after the execution of every statement. Like our approach, the extend timed Hoare logic introduced is also base language independent.

In this framework, timing specifications of a program P are expressed in terms of partial correctness assertions of the form $\{P\}S\{Q\}$ where P and Q are predicates on the program variables and the clock variables. Informally, $\{P\}S\{Q\}$ states that whenever the invariant P is true before S is executed, then Q must be true after the successful execution of S.

To deal with uncertainties in timing behavior, for each statement S its execution time *interval* $T(S) = [t_{min}(S), t_{max}(S)]$ is computed. Here, $t_{min}(S)$ and $t_{max}(S)$ stand respectively for the lower- and upper-bound of S's execution time. The function T is defined inductively on the structure of the statement S. For example,

$$\begin{aligned}
T(S_1; S_2) &= T(S_1) + T(S_2) \\
T(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2) &= [t_{min}(B) + t_{min}(\textbf{then/else}) + \min(t_{min}(S_1), t_{min}(S_2)), \\
&\quad\; t_{max}(B) + t_{max}(\textbf{then/else}) + \max(t_{max}(S_1), t_{max}(S_2))]
\end{aligned}$$

where addition on time intervals $[a, b]+[c, d]$ is defined as $[a+c, b+d]$, and $t_{min}(\textbf{then/else})$ and $t_{min}(\textbf{then/else})$ are respectively the lower- and upper-bound on the execution times of branching.

In this extension, a Hoare logic assertion of the form $\{P\}S\{Q\}$ is reinterpreted as the assertion

$$\{P\}\langle S; RT := RT + T(S); \rangle\{Q\}$$

where $RT = [rt_{min}, rt_{max}]$ is the lower- and upper-bound on the real-time clock respectively. The precondition P and postcondition Q can make use of RT to specify timing constraints on the program. Local clock variables can be introduced in order to keep track of start time of statement. However, the convention is that only the global clock RT is continuously incremented. This protocol is similar to the convention of clock variables in timed-automata[3].

In order to deal with periodic behavior in loops such as **while** B **do** S **end while**, it is often necessary to make references to the iteration counts of loops. This need is satisfied in the framework by first transforming the above statement into:

$$n := 0;\ \textbf{while } B \textbf{ do } S; n := n + 1 \textbf{ end while}$$

where n is an auxiliary *counting variable*. Since n is not part of the original program's variables, both statements $n := 0$ and $n := n + 1$ will take zero time to execute. The semantics and usage of counting variables and TimeC's counter variables are identical.

The expressive power and complexity of Shaw's extended Hoare logic is in sharp divergence with our TimeC's, however. Since this framework is an extension of Hoare logic, it is possible to specify the *functional* invariants in the same predicate where the *timing* invariants are specified, leading to great expressive power but also complexity; contrast this with TimeC's approach where only the simplest timing constraints on programs can be specified.

Hoare logic is undecidable due to the presence of the consequence rule:

$$\frac{\models (A \Rightarrow A') \quad \{A'\}S\{B'\} \quad \models (B' \Rightarrow B)}{\{A\}S\{B\}}$$

Thus in order to make use of the extended timed Hoare logic in program verification, it is necessary to utilize non-trivial theorem proving tools. It is also unclear how program synthesis can be directly automated from such a powerful logic. In contrasts, due to its simplicity, program synthesis in TimeC can be applied using more "tractable" combinatorial techniques.

Another big difference philosophy between Shaw's approach and ours is that in Shaw's the timing function T used to estimate the execution time bounds is defined in terms of the constructs of the high level language. However, in order for this to be feasible, this approach must impose a particular high-level to low-level instruction mapping for all high level constructs. In contrast, in TimeC the bounds on execution time are computed on the low-level IR using more detailed machine architectural descriptions; the details of high level to low level mapping is unconstrained as long as this phase preserves the timing constraints faithfully.

7.5 Timed Temporal Logics

Many extensions to *Temporal Logic* were proposed in order to capture real-time constraints. We refer the readers to the excellent survey [4] where many of these logics are described and compared.

The main design decisions that lead to this large variety of logics are:

- Whether the underlying semantics is *linear* or *branching*.
- Whether the time domain we should be *discrete* (the natural numbers) or *dense* (the reals).
- The precise syntax by which the timing constraints are expressed.

Since our framework in this paper is linear (the semantics of annotated programs being presented as *sequences* of timed-stamped states) and the time domain is discrete, we will only consider for comparison the timed temporal logics that are compatible with these parameters. Additional logics and their comparative power are surveyed in [4].

In many respects, the comments made while comparing RTL to TimeC also apply to the comparison of any of the timed temporal logics described below with TimeC. Namely, all of these logics are strictly more expressive than TimeC. In particular, they can all express *liveness* properties such as

Every p must be followed by a q within at most 5 time units,

which cannot be expressed within TimeC.

To illustrate the syntax of each of these logics, we will show how they would express the two main properties that are the most frequently used in a TimeC specification:

Conditional Upper Bound: If there is a q following a p, then q occurs no later than 5 time units after the occurrence of p. This property can be expressed within TimeC by the following constraint

$$\mathsf{time}(q) - \mathsf{time}(p) \leq 5$$

Lower Bound: No q can occur within 3 time units after the occurrence of p. This property can be expressed within TimeC by the following constraint

$$\mathsf{time}(q) - \mathsf{time}(p) \geq 3$$

For simplicity, we assume that p and q can occur at most once in every computation.

7.5.1 The Timed Logic MTL

The logic MTL (*metric temporal logic*) [6] extends conventional temporal logic (TL) [24] by introducing a bounded version of the temporal operators which subscripts a temporal operator such as $\Box$ or $\Diamond$ by an interval $[l, u]$. The meaning of such an annotation is that the scope of the subscribed operator is limited to this interval. For example, the two canonical constraints mentioned above can be expressed by the following MTL formulas:

Conditional Upper Bound: $(p \wedge \Diamond q) \quad \Longrightarrow \quad \Diamond_{[0,5]} q.$

This formula states that, from the time p occurs and q's future occurrence is guaranteed, q should occur within at most 5 time units.

Lower Bound: $p \quad \Longrightarrow \quad \Box_{[0,3]} \neg q.$

This formula states that, no q can occur within 3 time units from an occurrence of p.

The use of bounded temporal operators has been advocated by Koymans, Vytopil, and de Roever ([21], [20], [19]), although an early proposal by Bernstein and Harter can be viewed as a precursor [8].

7.5.2 The Timed Logic TPTL

The logic TPTL (*timed temporal logic*) [5] extends conventional TL by adding special variables that can be viewed as recording the value of an assumed master clock at given instances. These are often described as *freeze variables* since they freeze the current value of the master clock for future comparison.

For example, the two canonical constraints mentioned above can be expressed by the following TPTL formulas:

Conditional Upper Bound: $\Box x.\Big((p \wedge \Diamond q) \quad \rightarrow \quad \Diamond y.(q \wedge y \leq x+5)\Big).$

This formula freezes the reading of the master clock in variable x as soon as an occurrence of $p \wedge \Diamond q$ is detected. It then predicts an occurrence of q at a future time y, such that $y \leq x+5$.

Lower Bound: $\Box x.\Big(p \quad \rightarrow \quad \Box y.(q \rightarrow y \geq x+3)\Big).$

This formula states that, every occurrence of p at time x and a following occurrence of q at time y must satisfy $y \geq x+3$, guaranteeing a time separation of at least 3 time units.

As has been shown in [5], when we consider the discrete time domain, TPTL and MTL have the same expressive power. However, when we move to dense time domains, TPTL becomes more expressive than MTL. For example, the formula

$$\Box x.(p \quad \rightarrow \quad \Diamond(q \wedge \Diamond y.(r \wedge y \leq x+5)))$$

states the property that

> Every stimulus p is followed by a response q and, then, by another response r, all within 5 time units of the stimulus p.

It is shown that this property cannot be expressed by any MTL formula over dense time domains.

7.5.3 The Logics RTTL and XCTL

The logic RTTL (*real-time temporal logic*) [26] uses a special variable T to represent the current value of the master clock, and allows arbitrary first-order quantification over rigid variables that record previous readings of the clock.

For example, the two canonical constraints mentioned above can be expressed by the following formulas:

Conditional Upper Bound: $\forall x.\Box\Big((p \wedge T = x \wedge \Diamond q) \quad \rightarrow \quad \Diamond(q \wedge T \leq x+5)\Big).$

This formula freezes the reading of the master clock in variable x as soon as an occurrence of $p \wedge \Diamond q$ is detected. It then predicts an occurrence of q at a future time, such that $T \leq x+5$.

Lower Bound: $\forall x.\Box\Big(p \wedge T = x \quad \rightarrow \quad \Box(q \rightarrow T \geq x+3)\Big).$

This formula states that, if p occurred at time $T = x$ then any following occurrence of q must satisfy $T \geq x + 3$, guaranteeing a time separation of at least 3 time units between any p and any following q.

A simpler version of this logic which does not allow quantification but allows arbitrary addition between time recording variables has been introduced in [13] under the name XCTL (*explicit clock temporal logic*). It is based on the observation that a formula is valid iff its universal closure is. Thus, if we remove the $\forall x$ quantification from the two RTTL formulas presented above, we obtain two corresponding XCTL formulas which express the same two canonical properties.

We can summarize the discussion of these timed logics by the statement that, over the discrete time domain, all three logics have equal expressive power. That is,

$$\text{MTL} \quad = \quad \text{TPTL} \quad = \quad \text{RTTL}$$

The logic XCTL is incomparable with these three. On one hand, since it does not allow explicit quantification over time-recording variables, there are simple timing properties it cannot express. On the other hand, due to its ability to add time-recording variables in an arbitrary way, it can express properties that cannot be expressed in any of the other logics.

8 Conclusion

In this paper, we presented a constraint language TimeC for specifying timing relationships in programs. Our language is aimed at enabling the rapid prototyping of embedded applications in the presence of compiler optimizations. Several interesting research issues emerge from the work summarized in this paper. Here, we will enumerate a short list of these issues that we are pursuing, in the context of our own research effort.

1. Engineering TimeC and Time_tract constraints into an optimizing compiler environment and validating their use.

2. Design of instruction scheduling optimizations that are sensitive to timing constraints, in addition to conventional program dependences.

3. Currently, instruction scheduling optimizations do not assume knowledge of the program's dynamic behavior. However, the art is moving closer towards what is commonly referred to as *profile-driven optimization* wherein a statistical model of the program's execution is assumed to be known. Attributes such as branch frequencies, the probability of executing two instructions together, the probability of two apparently different addresses being aliased together are examples.

   There is a lot of potentially very interesting and useful work that can be performed, by adapting these profile-driven or profile-guided optimizations, to cope with timing constraints.

4. Finally, we believe there is plenty of room for further development based on our constraint notation. Instead of using a formalism derived from, say, temporal logic, we have chosen to use a notation that is simpler and thus less daunting for typical developers of embedded applications. One suggested direction for future work could utilize two levels of constraints notation: the interaction level could resemble TimeC, which the users see. The level internal to the compiler could consist of constraints directly expressed in terms of the program's graph-based intermediate representation and thus be more amenable to analysis and transformation. These issues of usability and engineering will require further research.

In conclusion, we view TimeC as a catalyst for enabling end-to-end solution, starting with timing constraints and ending with executable code optimized by efficient algorithms that obeys these constraints, for embedded applications.

References

[1] R. Sethi A. Aho and J. Ullman. *Compiler Construction.* Addison-Wessley, 1984.

[2] A. Aiken and A. Nicolau. Loop quantization; an analysis and algorithm. Technical Report 87-821, Cornell University, March 1987.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] R. Alur and T. Henzinger. Logics and models of real time: A survey. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lect. Notes in Comp. Sci.*, pages 74–106. Springer-Verlag, 1992.

[5] R. Alur and T.A. Henzinger. A really temporal logic. In *Proc. 30th IEEE Symp. Found. of Comp. Sci.*, pages 164–169, 1989.

[6] R. Alur and T.A. Henzinger. Real-time logics: Complexity and expressiveness. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, pages 390–401, 1990.

[7] Intel and Hewlett Packard Corporations. Press releases on Intel's IA-64. `http://www.intel.com/pressroom/kits/events/mpf1097.htm`, 1997.

[8] A. Bernstein and P. K. Harter. Proving real time properties of programs with temporal logic. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 1–11. ACM, 1981.

[9] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[10] B.W.Kernigan and D.M.Richie. *The C Programming Language.* Prentice Hall, 2 edition, 1988.

[11] J. Fisher. Trace scheduling: A general technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[12] Joseph Fisher. Global code generation for instruction-level parallelism:trace scheduling-2. Technical report, HP Labs, 1991.

[13] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proc. 5th IEEE Symp. Logic in Comp. Sci.*, pages 402–413, 1990.

[14] C.A.R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12:576–580, October 1969.

[15] S. Hong and R. Gerber. Compiling real-time programs with timing constraint refinement and structural code motion. *IEEE Transactions on Software Engineering*, 21, May 1995.

[16] F. Jahnaian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Soaftware Engineering*, SE-12(9):890–904, September 1986.

[17] F. Jahnaian and A. K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C36(8), 1987.

[18] J.R.Allen, K.Kennedy, C. Porterfield, and J. Warren. Conversion of contgrol dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[19] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time Systems*, 2(4):255–299, 1990.

[20] R. Koymans and W.-P. de Roever. Examples of a real-time temporal logic specifications. In B.D. Denvir, W.T. Harwood, M.I. Jackson, and M.J. Wray, editors, *The Analysis of Concurrent Systems*, volume 207 of *Lect. Notes in Comp. Sci.*, pages 231–252. Springer-Verlag, 1985.

[21] R. Koymans, J. Vytopyl, and W.-P. de Roever. Real-time programming and asynchronous message passing. In *Proc. 2nd ACM Symp. Princ. of Dist. Comp.*, pages 187–197, 1983.

[22] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. *Proceedings SIGPLAN'88 Symposium on Programming Language Design and Implementation*, pages 318–328, 1988.

[23] A. Leung, K. Palem, and A. Pnueli. Scheduling instructions with real-time constraints on modern microprocessors. *Unpublished*, 1998.

[24] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, London, 1981.

[25] N. Nachiappan. Personal communications and memorandum of support, 1995.

[26] J.S. Ostroff. *Temporal Logic of Real-Time Systems.* Advanced Software Development Series. Research Studies Press (John Wiley & Sons), Taunton, England, 1990.

[27] K. Palem and V. Sarkar. *Code Optimization in Modern Compilers.* Western Institute of Computer Science, Stanford University, CA, 1995.

[28] B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. *Proceedings of the 27th Annual Symposium on Microarchitecture*, December 1994.

[29] Alan Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[30] Alexander D. stoyenko, Thomas J. Marlowe, and Phillip A. Laplante. A description language for engineering of complex real-time systems. Technical Report cis9522, New Jersey Institute of Technology, 1996.

[31] Alexander D. stoyenko, Thomas J. Marlowe, and Mohamed F. Younis. A language for complex real-time systems. Technical Report cis9521, New Jersey Institute of Technology, 1996.

[32] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2 edition, 1990.

[33] Nancy Water, Scott Mahlke, Wen mei Hwu, and B. Rau. Reverse If-conversion. In *ACM SIGPLAN PLDI*, pages 290–299, 1993.