# Language Constructs for High-Precision Specification of Real-Time Constraints *

Tai M. Chung and Hank G. Dietz

School of Electrical Engineering

Purdue University

West Lafayette, IN 47907

tai@ecn.purdue.edu, hankd@ecn.purdue.edu

**Abstract**

In practice, the hard real-time systems are still implemented in low-level programming languages for time critical portions and manually tuned to meet all the timing requirements. Without a real-time language that supports an appropriate way of specifying time constraints for a generic hard real-time systems and the fine-grain timing analysis that is transparent to users, the users will ever suffer from the complex coding and accurate analysis, particularly for the systems requiring fast turn-around responses.

In this paper, we propose novel language constructs that can be added to any imperative programming language so that the extended language provides users a way to specify relative time constraints between arbitrary operations at instruction-level. The compilation techniques unique to transformation of proposed language are also presented as a part of the CHaRTS, Compiler for Hard Real-Time Systems, that generates a valid instruction sequence for a target execution model.

**Key words:** hard real-time, language, time constraint, CHaRTS

*This work is supported by the Office of Naval Research (ONR) under grant number N00014-91-J-4013.

# 1 Introduction

In our view of real-time systems, a real-time system consists of a controlling subsystem and controlled entities. A controlling subsystem is a set of computer systems, while the controlled entities can be any of a broad range of systems with mechanical behavior, any device from a simple blender to a complex robot [SR91] [BHJ+82]. Typically, a controlling subsystem executes control programs to receive input from the environment and/or to send commands to the controlled entities appropriately.

For a real-time system to function correctly, the control program must be logically correct and the controlling subsystem must execute the program without timing faults. Either a failure to perform an action at the appropriate time or a flaw in the control program's logic can yield catastrophic consequences in *hard real-time* systems. Thus, meeting the time constraints is extremely important in such systems.

Based on when the control program is scheduled, real-time systems can be divided into two categories: dynamic and static. Execution order of control tasks in a dynamic system is determined on-the-fly at run-time by a scheduler that examines the current status of the system; often, this scheduler is part of the operating system. Even though the dynamic systems are flexible, they suffer from scheduling overhead and unpredictable risks.

In contrast, a static system is scheduled at compile time based on analysis of the program, time constraints, and resource use predictions [Loc92]. A static system schedules the execution order at compile time based on the predicted behavior of the controlled entities and the timing properties of the controlling subsystems. Thus, the static system guarantees that properly scheduled code will function without a timing failure. However, despite the fact that guaranteed timely execution makes the static systems much more attractive than dynamic systems in a hard real-time environment, static systems have received less attention than dynamic systems. This is due to:

1. Unpredictable machine behavior: general-purpose processors often implement instructions with execution time dependent on operand values, pipeline conflicts, or memory hierarchy use (e.g., cache misses, dynamic RAM refresh cycles, virtual memory page faults). Any of these variables degrades the accuracy of the required timing analysis.

2. Scheduling complexity: both the instruction-level timing analysis and the code reorganization are NP-complete problems [GJ79]. Thus, static scheduling has focused on coarse-grain tasks to reduce the problem size.

3. Lack of programming support: no programming language supports a mechanism to express fully general real-time constraints.

The first problem is not easily solved, but can be avoided by careful design of the computer hardware. For example, the TMS320C40 high-performance DSP has completely static timing properties, provided that interrupts are disabled and local memory is implemented by SRAM [Inc88]. Even interactions between multiple processors can be made to have static timing properties. For example, PAPERS (Purdue's Adapter for Parallel Execution and Rapid Synchronization) can provide fine-grain communication and synchronization with precise static timing properties [DMSM94] [DCMM95].

Although further work is needed, we believe that the techniques presented in [CD95] form the basic foundation for an appropriate static scheduling algorithm. Thus, the second problem is partially solved and a practical solution is likely to be found soon.

This leaves the problem of providing an appropriate programming model and language. Part of this problem is that specifying fine-grain time constraints in the context of a high-level programming language seems paradoxical. Assembly language hand coding to resolve fine-grain time constraints is not the answer; such programs are difficult to write and maintain, and automated scheduling is almost impossible [Inc86].

It is equally futile to directly use existing high-level real-time programming languages, such as ADA [Bre81], Edison [Han80], or programming languages extended from the general purpose languages like RT-Euclid [KS86], FLEX [LN88], MPL [NTA90] and TCEL [GH93], because they do not provide a mechanism for specifying time constraints on any language construct finer in granularity than an entire task. These languages also suffer from use of a programming model that represents time constraints as relationships between task pairs that are lexicographically adjacent in the source program; this adjacency constraint is both artificial and insufficiently expressive. Timing relationships between arbitrary (potentially parallel) operations cannot be expressed because some such relationships cannot be mapped into lexicographically adjacent positions in the program.

Section 2 of this paper proposes novel language constructs that can be added to any imperative programming language so that the extended language provides users with a way to specify relative time constraints between arbitrary actions at the level of individual instructions. To aid the reader in understanding how these time constraints are extracted and processed, Section 3 provides a brief summary of the relevant compiler technology. Section 4 summarizes the contributions of this paper and suggests future directions for research.

## 2 Real-time language

To motivate the design of the new language constructs, consider the informally annotated control program fragment shown in Figure 1. Although this example is remarkably simple and embodies time constraints that could easily have been found in a real application, none of the existing programming languages is capable of expressing this in a form that would facilitate static timing analysis and scheduling.

```
i1 : Load r1,sense1 ; Get value from memory-mapped sensor 1
                    ; i1 must execute no later than 1 µs after i3
i2 : Store act1,r1  ; Send value to actuator 1
                    ; i2 must execute at least 1 µs after i5
i3 : Load r2,sense2 ; Get value from memory-mapped sensor 2
                    ; i3 must execute no later than 1 µs after i1
i4 : Add r3,r1,r2   ; Add the sensor inputs
i5 : Store act2,r3  ; Send result to actuator 2
```

Figure 1: An example of code segment for real-time control actions

### 2.1 Programming model

The real-time language we propose in this paper implements a graphical model, denoted as $G = (\Gamma, \Theta)$ where the directed graph $G$ consists of a set of nodes $\Gamma$ and a set of edges $\Theta$. The nodes and edges are associated with instructions and real-time constraints respectively. The graphical model, called a directed timed graph or DTG, is distinguished from conventional directed graphs in several aspects. Most importantly, each edge in a DTG does not necessarily indicate a precedence or dependence relation, but represents the direction of the temporal relation between two control action.

An instruction in $\Gamma$ can be classified as an *externally viewed instruction*(EVI) or an *internally viewed instruction*(IVI) based on the effect of the instruction. The effect of IVI is limited to the internal computation, while an EVI changes the status of the control environment. For example, the variables defined as *volatile* in the C language [X3.89] or commands to control robots using RCCL [LHHL88] are EVIs, and their execution must meet the timing constraints specified. Because EVIs may depend on values computed by IVIs, any such dependence also implies a relative time constraint (execution order) which must be preserved.

A directed edge in $E$ is associated with multiple attributes: source object $\gamma_s$, sink object $\gamma_e$, relational operator $\eta$, and offset $\delta$. A constraint $\theta_k$ is defined as $\theta_k =< \gamma_s, \gamma_e, \eta, \delta >$. The type of relationship represented by each edge is specified by $\eta$: before($<$), after($>$), concurrent($=$), or exclusive($\neq$). Indeed, any temporal relations between $\gamma_s$ and $\gamma_e$ for each edge type can be expressed with time value $\delta$ as:

- before constraint : $\gamma_e < \gamma_s + \delta$ ($\gamma_e$ must happen at latest $\delta$ after $\gamma_s$)
- after constraint : $\gamma_e > \gamma_s + \delta$ ($\gamma_e$ must happen at earliest $\delta$ after $\gamma_s$)
- concurrent constraint : $\gamma_e = \gamma_s + \delta$ ($\gamma_e$ must happen exactly at $\delta$ after $\gamma_s$)
- exclusive constraint : $\gamma_e \neq \gamma_s + \delta$ ($\gamma_e$ must NOT happen at $\delta$ after $\gamma_s$)

An edge representing the direction of the temporal relation between the source instruction ($\gamma_s$) and sink instruction ($\gamma_e$) follows a simple rule. The $\gamma_s$ and $\gamma_e$ appear, respectively, in the RHS and the LHS of the constraint.

The $\gamma_s$ and $\gamma_e$ can be any arbitrary pair of instructions. Moreover, this method is sufficient to specify the constraints among $n$ instructions using less than *n(n-1)/2* constraints in this model. That is, any constraint among multiple instructions can be specified by our constraint model – even constraints which cannot be satisfied by a sequential schedule (i.e., constraints which imply parallel execution).

It is significant that simple ordering (precedence) constraints can be represented without introducing any additional types. Each precedence constraint of the form "*x uses y's result*" can be encoded as the *after* constraint "*x happens at earliest 0 after y*". Hence, both timing constraints and precedence constraints can be expressed using the same syntax.

## 2.2 Desired features in Real-time languages

The single most important aspect of any programming system is how easily that system can be used to create and maintain working programs. Although the relative ease of programming of various languages is largely a matter of "religious" debate, it is clear that familiar language constructs and programming tools are more effective than if a completely new program development style must be adopted. The real-time language should look and "feel" like a popular conventional language, e.g., C. Likewise, because there is no familiar syntax that can be borrowed for expressing time constraints, it is critical that the timing aspects of the code be directly observable and easy to modify. Any timing constraint involving two operations should be able to be expressed directly, without having to adjust other portions of the code.

Part of the ability to treat each constraint independently depends on the ability to combine multiple constraints that affect the same operation. Although most work on real-time systems suggests that such constraints are always combined by requiring *all* constraints to be met, it is sometimes necessary [GL91] to combine constraints by requiring that *any* one or more of these constraints be met. This concept of an *oring* constraint is quite new in the real-time community. Suppose an event $A_1$ can be execute either before $A_2$ or after $A_3$ as shown in Figure 2. This can be mathematically expressed as $A_1 < A_2$ or $A_1 > A_3$. The figure 2 shows that the possible range of $A_1$ could be different for the same constraint based on the temporal values of $A_2$ and $A_3$. Despite the usefulness and necessity of such a construct, *oring* constraints have been conspicuously absent from other real-time programming languages, perhaps because the scheduling is made more complex by supporting both types of constraint combining.

We further suggest that it is critical that the timing constraints be expressed at the finest possible grain level. Because finer grain yields more freedom in the compiler's scheduling, this yields the highest probability that all timing constraints can be met. Of course, more freedom in scheduling also implies a larger search space for schedules, but the user does not need to be aware of the additional complexity in the compiler.

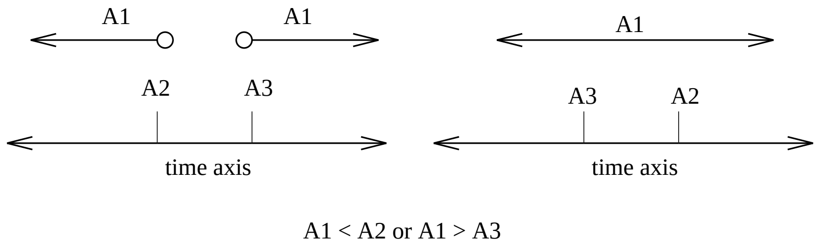


Figure 2: An example of *oring* constraint

The concept of a cyclic task in a program is a structure that is unique to real-time systems, but is commonly used. There is no obvious way to obtain the same effect using other constructs, and subtle methods destroy maintainability. Thus, we suggest that cyclic structures should be directly represented in the language, making them easy to recognize for both programmer and compiler. In fact, the language should allow multiple arbitrary cyclic tasks to be specified without concern for how the different length cycles will be converted into a single coherent schedule.

## 2.3 Parent language

Thus far, none of the existing real-time languages provides all the features mentioned above. Most of the real-time languages are not easy to program, not based on fine-grain tasks, and/or not designed to express relative constraints among arbitrary control operations. Thus, we propose an extension of an imperative language to facilitate the features. Even though any imperative language can be extended for those features, we develop novel language constructs onto a subset of the C language because C is used for most of the embedded real-time systems in conjunction with assembly macros for time critical portions.

The parent language we propose includes all expressions using arithmetic and relational operators, **if** statement, and **while** statement. At this time, neither aliasing nor floating point computation is supported.

## 2.4 Proposed language constructs

In order to specify the constraints on top of the parent language, we introduce several new constructs: *timing block*, *temporal expression* and *cyclic block*. A timing block defines a code segment to be manipulated as a unit for the purpose of constraint specification, while time expression specifies the relative constraints in terms of the timing blocks. A cyclic block is defined for the code sequence that runs indefinitely or until the program terminates. A program may contain more than one cyclic block.

Before illustrating the constructs, consider an example of a control program shown in Figure 3. The program is written to read data from memory locations where they are updated by the devices, and to write commands to memory locations where they are dispatched by a to the devices at desired times.

### 2.4.1 Timing block

A timing block is a user defined variable or a sequence of statements that could be viewed as a task in the traditional real-time systems. That is, it contains either a volatile variable (EVI) or statement(s) that consist of at least one EVI. If a timing block contains no EVI, the temporal expressions associated with the temporal variable are ignored, implying that the constraint is automatically satisfied.

The timing variable identifying a timing block appears between "@" and ":" that is followed by an EVI or a body of the block as shown in the syntax;

```
@tvar:    variable
or
@tvar:    { (statement)+ }
```

Here, `tvar` is a user defined name for the block and `variable` is a regular variable that is an EVI [1]. The notation `(statement)+` denotes one or more statements composing a timing block. For example, line 12 in Figure 3 defines temporal variables `tv2` and `tv3` that are associated with a variable `sensor2` and a statement `b = sensor2 + 4;`. In this case, `tv2` is a nested block of `tv3`.

[1]The volatile variables are translated into one or more instructions in low-level languages

```
main()
{
    volatile char action1, action2, action3;
    volatile int sense1, sense2, sense3;
    char command1, command2;
    int a,b,c,d;

    d = 23;
    cycle @tp1:(5) {
        /* Read in data which should be stored by actual sensors.  */
        a = @tv1:sensor1 + 5 + d;
        @tv3:{ b = @tv2:sensor2 + 4;}
        /* Specifying the timing between the reads */
        @:  tv1.  < tv2.  + 1 | tv2.  < tv1.  + 1;
        @:  .tv3 < tv2.  + 3;
        /* Action is performed based on the data read */
        if(a < b)
            @tv4:{action1 = command1;}
        else
            @tv5:{action2 = command2;}
        /* Specifying the timing for action */
        @:  tv22 = tv2 + 13;
        @:  .tv4 > tv22.  + 1;

        /* Also specifying the timing over the iterations */
        @tv6:{
            while( a < b ) {
                @tv61:c = a + b - 23;
                c = c * 23 - foo();
                a = a + 1;
        @:  .tv6[2] < tv6[1].  + 25;
        } }
        @tv8:{action2 = command1 + c;}
        @:  .tv8 > .tv6 +10;
    }

    cycle @tp2:(10) {
        int c; c = 43;
        d = @vt11:sensor3 + sense1;
        @tv12:{action3 = d * c + 3;}
        @:  vt12.  < vt11.  + 14;
    }
    /* The first action of cycle tp2 should be started
    before the the last ection of cycle tp1 is completed */
    @:  .tp2 < tp1.  + 2;
    exit(0);
}
```

Figure 3: Example of A Control Program With New Constructs

A timing block has all the properties of regular block, such that it is lexically scoped and variable declarations are allowed in the beginning of the block. Also, the timing blocks can be nested unless they are interleaved. The interleaving of the timing blocks is disallowed because it increases the complexity and destroys the structured language feature.

### 2.4.2 Cyclic block

Cyclic block is a special case of timing block in the sense that the block is executed repeatedly until the program is terminated. Thus, compiler should treat this block differently from other blocks for analysis as well as scheduling. A cyclic block is identified by a temporal variable `tvar` and the period $\delta$ as defined here.

**cycle** `@tvar:(`$\delta$`) { statement(s) }`

The **cycle** is a keyword indicating that the following block enclosed by { and } is a cyclic block, `tvar` is a temporal variable name for the block, and $\delta$ specifies the timing requirement that the cyclic block must complete within. That is, the `statements` in the timing block and looping overhead must be completed within $\delta$.

The time constraints associated with a cyclic block may be expressed in the temporal expressions described later. For example, the program in Figure 3 consists of two cyclic blocks `tp1` (line 9 to 35) and `tp2` (line 37 to 42), and a temporal relation between the cyclic blocks is specified in line 45. The temporal expression in line 45 implies that the first action of `tp2` should not be executed later than 2 time units after the last action of `tp1`.

### 2.4.3 Temporal expression

A temporal expression is classified as either a temporal assignment or a temporal relation. A temporal assignment is a way of defining another temporal variable that has temporal distance from an existing temporal variable, while a temporal relation is used to express the relationship between the temporal variables.

`@: tvar = tvar (+ | -)` $\delta$`;`

or
@: etvar $\eta$ etvar (+ | -) $\delta$ ( | etvar $\eta$ etvar (+ | -) $\delta$)*;

In this syntax, `tvar` is the name of a temporal variable, and `etvar` is extended temporal variable indicating start time or completion time using prefixed-dot or postfixed-dot respectively. In Figure 3, `tv1.` in line 14 indicates the completion time of `tv1` while `tv3.` in line 15 indicates the start time of `tv3`. Also, $\eta$ is one of the temporal relation operator defined in the previous section, and $\delta$ is the offset that is any time unit as small as a clock cycle. Also, the temporal relation ( | etvar1 $\eta$ etvar2 (+ | -) $\delta$)* indicates zero or more temporal relations to specify *oring* constraint.

As we have seen in the syntax of the temporal assignment, computation upon the temporal variables is allowed. For example, temporal assignment `tv2 = tv1 + 7` and temporal relation `.tv3 < .tv2 + 7` together implies that the start time of `tv3` must happen earlier than 14 time units after the start time of `tv3` which is equivalent to `.tv3 < .tv1 + 7 + 7`.

The Figure 3 illustrates one unique temporal expression for the time constraints involving the iterations. When a block that appears in iterated loop is considered, it is necessary to specify time constraints between the instances in different iterations. We introduce array of temporal variable to resolve this problem. As we see in line 31 of Figure 3, the distance between the instances are expressed as the index of the array. For example, line 31 says that the loop should be executed within 25 time units because the temporal expression requires that the first action of $(i+1)$th iteration should be started earlier than the 25 time units after last action of $i$th iteration. At this point, we formally define `etar` as;

etvar : . tvar {[i]} | tvar . {[i]}

where `tvar` is a temporal variable and {[i]} is an optional expression of `[i]` denoting `ith` instance of the iteration.

### 2.4.4 Illegal expressions

The lines 17 to 20 of the Figure 3 defines the temporal variables `tv4` and `tv5` and lines 22 to 23 shows corresponding temporal expressions, temporal assignment and temporal relation. It says that the action in *then* clause should be performed later than 14 time units after `tv2` is completed. Thus, the timing constraints is taken into account only when `a < b` holds. However, any temporal expression who has a pair of instructions $\gamma_s$ and $\gamma_e$, one from *then* clause and the other from *else* clauses are illegal because it is impossible to run both branches in an execution.

Another illegal expression is the statement whose pair of instructions necessarily include a statement with unpredictable timing property between them. Those statements are any instructions whose timings are not predictable such as *break* in a loop, unpredictable number of iterations, and unpredictable instructions involving interrupts. For example, the execution of `tv6` (line 26 to 35) is repeated as long as $a < b$ holds. Hence, if the range of $a$ and $b$ are non-deterministic, the temporal expression in line 34 is an illegal expression.

Temporal expression specifying a timely conflict actions is also classified as an illegal expression. Consider two expressions in line 13 and 14, and modify line 13 to `tv3. > .tv2 + 4`. Then, `tv3. > .tv2 + 4` and `.tv3 < tv2. + 3` has no solution, implying that the code cannot have valid schedule. Thus, those two temporal expressions are illegal. These illegal expressions are found only by thorough semantic analysis.

# 3 Compilation techniques

In this section, we briefly explain the compilation techniques unique to process the language transformation. Particularly the compiler organization, data structure of intermediate representation, and cyclic block scheduling are focused because they are either very much different from conventional compilation techniques [ASU86] [Die87] or novel concept to the compilation.

## 3.1 Compiler organization

As we see in Figure 4, CHaRTS consists of four software modules; syntax analysis, semantic analysis, scheduling, and code generation module. Each module is designed and implemented to be independent so that the software can be individually replaced for enhancement. For example, the scheduling algorithm we developed [CD95] can be replaced with better one without affecting any other modules except minimal modification on interface between them. In our implementation, those modules are written in the C language with the support of PCCTS (Purdue Compiler Construction Tool Set).

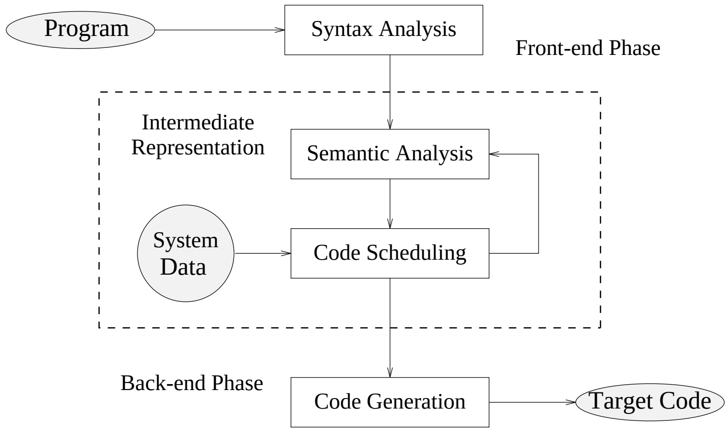


Figure 4: Compiler Architecture for Hard Real-Time Language

The syntax analysis module performs building an intermediate representation from a source program. The intermediate representation of CHaRTS is unique as described in section 3.2 because of the timing constraint specification and cyclic blocks.

The function of the semantic analysis module is to verify a program context against the language semantics including;

- Does the timing block contain at least one EVI?
- Is the temporal expression semantically legal?
- Is it compliant to the parent language?

The objective of the code scheduling module is to reorganize the instruction sequence to meet all the timing constraints as well as the logical correctness. This module and code generation module are entirely system dependent and the timing properties of the hardware components and the instruction sets can be either hard-wired into the compiler or read from system file as shown in the Figure 4. The input to the scheduling module is the tuples whose timing characteristic is close to the object code so that the execution time of the tuples are preserved at code generation time.

### 3.2 Intermediate representation

In addition to the traditional role of a lexical analyzer and parser, the syntax analysis module extracts constraints and information of timing blocks from the program and builds an intermediate representation. As depicted in Figure 5, the data structure of the intermediate representation consists of two tables and three trees; The tables are a symbol table (SymTab) and a table for temporal variables (TVarTab), while the tree structures include an abstract syntax tree (AST), a cycle syntax tree (CST), and a temporal syntax tree (TST) that are all child-sibling trees [PDC92].

Although only AST and SymTab are sufficient data structure in conventional compilers, the compiler for real-time systems we propose needs more information, and they are organized in TVarTab to store information of the timing blocks such as block type, entry point, and exit point, CST to manipulate all the blocks to be executed periodically, and TST to represent the temporal expressions.

In particular, manipulation of CST in conjunction with temporal expressions specifying the timing constraints among the cyclic blocks is quite complex and is new concept to the compiler community because multiple cyclic blocks need to be executed indefinitely, considering the period and timing constraints among the cyclic blocks.

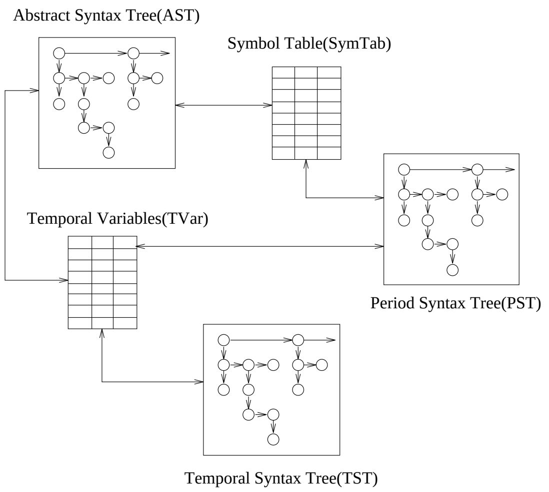


Figure 5: Data Structure in Intermediate Representation of CHaRTS

## 3.3 Cyclic block scheduling

Scheduling the cyclic blocks is very different from the conventional compilation techniques because conventional compilation does not have a notion of interleaving loops. The cyclic blocks need to be integrated into a single cyclic block that can be executed without an explicit dynamic scheduler or a dispatcher. In this section, we develop a novel scheme to integrate the given cyclic blocks, associated with different periods, into a single cyclic block. The integrated cyclic block executed indefinitely or semi-indefinitely until the program terminates.

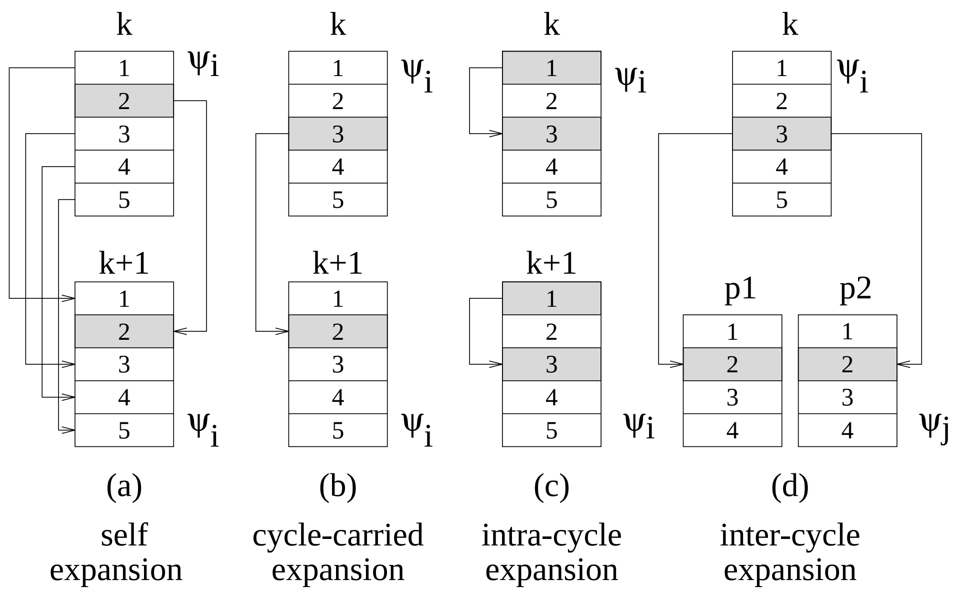


Figure 6: Cyclic block expansion of the timing constrains

Let $\psi_1, \psi_2, \cdots, \psi_n$ be the cyclic blocks associated with the periods $p_1, p_2, \cdots, p_n$; that is, the execution of the first temporal variable in block $\psi_i$ should be started every $p_i$. The re-arrangement of the cyclic blocks into a single cyclic block $\Psi$ is processed as follow using the fundamental property of scheduling of the schedule for a periodic task [LM80].

1. compute $p = \mathcal{LCM}_{i=1}^{n}(p_i)$.

2. $\forall i$, replicate the instructions in $\psi_i$ for $p/p_i$ times.

3. $\forall i$, expand the timing constraints. Notice that the expansion is a simple addition of temporal

expressions in our model because *dependence* is a subtype of simple constraint. Define $\gamma^k_{i(x)}$ as the instruction $x$ in $k_{th}$ instance of $\psi_i$, say $\psi^k_i$, and $\psi_i$ as the original cyclic block associated with period $p_i$. Likewise, $\gamma_{i(x)}$ represents the instruction $x$ in $\psi_i$. The constraint expansion is performed based on the following analysis.

- **self expansion:** The replicated instructions add timing constraints to maintain the execution order of those replicated instructions. At this expansion, the periodic constraint is enforced on the first control action. Namely, $\forall i$ and $\forall j$, a temporal expression $\gamma^k_{i(x)} < \gamma^{k+1}_{i(x)} + \delta$ is added where $\delta = p_i$ if $\gamma_x$ is the first control action in $\psi_i$ or $\delta = 0$ otherwise [2]. The graphical representation of self-expansion is depicted in (a) of Figure 6 when instruction 1, $\gamma^k_{i(2)}$ is the first control action in $\psi_i$.
- **cycle-carried expansion:** Suppose $\gamma^{k+1}_{i(x)}$ is dependent on $\gamma^k_{i(y)}$, say $\gamma^k_{i(y)} \prec \gamma^{k+1}_{i(x)}$ for $k > 0$. In this case, the dependence is enforced for every replicas of $\psi_i$, adding temporal expressions of $\gamma^k_{i(y)} < \gamma^{k+1}_{i(y)}$ for $1 \leq k \leq n-1$. This expansion is applied to every cyclic carried dependences and shown in (b) of Figure 6. In the figure, $\gamma^{k+1}_{i(2)}$ uses the result of $\gamma^k_{i(3)}$; thus, the temporal expression $\gamma^k_{i(3)} \prec \gamma^{k+1}_{i(2)}$ for $k > 0$.
- **intra-cycle expansion:** Temporal expressions specifying the constraints between the instructions in the same block are replicated as necessary. That is, for $\gamma^0_{i(x)} < \gamma^0_{i(y)} + \delta$, $\gamma^k_{i(x)} < \gamma^k_{i(y)}$ is added in every replicas of $\psi_i$. In (c) of Figure 6, the temporal expression $\gamma^k_{i(3)} > \gamma^k_{i(2)} + \delta$ is replicated in $\psi^k_i, \forall k$.
- **inter-cycle expansion:** Temporal expressions specifying the constraints between the instructions beyond one block (not within a block) are replicated as necessary. Let $\xi^k_{i(x)}$ be a starting time of $\gamma^k_{i(x)}$ in $\psi^k_i$, Then, the range of $\xi^k_{i(x)}$ can be expressed as $(k-1)\times p_i \leq \xi^k_{i(x)} \leq (k+1)\times p_i$. Thus, the inter-cycle temporal expression $\gamma_{i(x)} < \gamma_{j(y)}+\delta$ yields the temporal expressions $\gamma^k_{i(x)} < \gamma^p_{j(y)} + \delta, \forall p$ such that $\psi^p_j$ satisfies $\xi^k_{i(x)} \wedge \xi^p_{j(y)} \neq \emptyset$. Consider (d) of Figure 6. When we assume that $p = p1$ or $p2$ for $\xi^k_{i(2)} \wedge \xi^p_{j(1)} \neq \emptyset$, temporal expressions $\gamma^k_{i(3)} < \gamma^{p1}_{j(2)} + \delta$ and $\gamma^k_{i(3)} < \gamma^{p2}_{j(2)} + \delta$ are added.

4. $\Psi$ is scheduled by an instruction level fine-grain scheduling algorithm. One of the approaches to find a valid schedule by using a genetic search based algorithm is given in [CD95].

[2]We assume that the $p_i$ implicates that the first control action in $\psi_i$ should be repeated every $p_i$. The other instructions in $\psi_i$ are correctly executed as long as the expanded dependences are not violated.

## 4 Conclusion

Building a statically-scheduled real-time system is not as difficult as it first appears – if we have an appropriate real-time language and compiler technology. After making significant strides toward development of the compiler technology, we believe that the key to building the language is providing a very general model which both the programmer and the compiler can easily understand and manipulate. This paper has proposed such a language.

Basic features of this language include:

- The basic syntax of a conventional language (in this paper, C) and compatibility with that language's programming idioms
- The ability to place relative time constraints on arbitrary pairs of fine-grain operations or on cyclic blocks
- The ability to specify arbitrary types of relative time constraints (e.g., any of *before*, *after*, *concurrent*, and *exclusive* combined by either *anding* or *oring*)

Some of these features greatly complicate the compiler's job by enlarging the schedule search space or by adding complexity to the evaluation of constraints. However, we have made good progress toward solving these problems, and nothing in the language lies far beyond the ability of the new compiler technology we have developed thus far.

Currently, the compilation techniques like the control structure analysis, the code scheduling with a hierarchical decomposition scheme, and the code generation for a parallel execution model (PAPERS) are under development. Indeed, current and future work centers on the implementation of a complete compiler for this language: CHaRTS (Compiler for Hard Real-Time Systems).

# References


[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compiler Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[BHJ+82] M. Brady, J. M. Hollerbachand, T. L. Johnson, T. Lozano-Perez, and M. Mason. *Robot Motion : Planning and Control.* MIT Press, Cambridge, Massachusetts, 1982.

[Bre81] R. F. Brender. What is ADA? *IEEE Computer*, 14(6):17 – 24, June 1981.

[CD95] T. M. Chung and H. G. Dietz. Adaptive genetic algorithm: Scheduling hard real-time control programs with arbitrary timing constraints. *to appear as Technical Report of EE at Purdue University*, May 1995.

[DCMM95] H. G. Dietz, T. M. Chung, T. I. Mattox, and T. Muhammad. Ttl_papers: An implementation of purdue's adapter for parallel execution and rapid synchronization. *submitted to 24th IEEE International Conference on Parallel Processing*, August 1995.

[Die87] H. G. Dietz. *The Refined-Language Approach to Compiling for Parallel Supercomputers.* PhD thesis, Polytechnic University, Brooklyn, New York, June 1987.

[DMSM94] H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox. PAPERS: Purdue's Adaptor for Parallel Execution and Rapid Synchronization. Technical Report TR-EE 91-11, Purdue University, March 1994.

[GH93] R. Gerber and S. Hong. Semantic-based compiler transformation for enhanced schedulability. In *IEEE Proc. of Real-Time Systems Symposium*, pages 232 – 242, December 1993.

[GJ79] M. R. Garey and D. J. Johnson. *Computers and Intractability, a Guide to Theory of NP-Completeness.* W. H. Freeman Company, San Francisco, CA, 1979.

[GL91] D. W. Gillibs and J. W. Liu. Scheduling tasks with and/or precedence constraints. Technical Report UIUC-ENG-1766, University of Illinois, October 1991.

[Han80] P. B. Hansen. Edison: A multiprocessor language. Technical Report Dept of Computer Science, University of Southern California, September 1980.

[Inc86] Unimation Incorporated. *Unimate Industrial Robot Programming manual, User's Guide to VAL II.* Unimation Incorporated, Danbury, Connecticut, December 1986.

[Inc88] Texas Instruments Inc. *Third Generation TMS320 User's Guide.* Texas Instruments Inc., 1988.

[KS86] E. Kligerman and A. D. Stoyenko. Real-Time EUCLID: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):941 – 949, September 1986.

[LHHL88] J. S. Lee, S. Hayati, V. Hayward, and J. E. Lloyd. Implementation of RCCL, a Robot Control C Library on a MicroVAX II. In *Intelligent Robots and Computer Vision*, pages 472 – 480, Cambridge, MA, Oct 1988.

[LM80] J. Y. T. Leung and M. L. Merril. A note on preemptive scheduling of periodic real-time tasks. *Information Science Letter*, 20(3):115 – 118, March 1980.

[LN88] K. J. Lin and S. Natarajan. Expressing and Maintaining Timing Constraints in FLEX. In *IEEE Proc. of Real-Time Systems Symposium*, pages 96 – 105, December 1988.

[Loc92] D. C. Locke. Software architecture for hard real-time applications: cyclic vs. fixed priority executives. *Real-time Systems*, 4(1):37 – 53, March 1992.

[NTA90] V. M. Nirkhe, S. K. Tritaphi, and A. K. Agrawala. Language Support for the Maruti Real-time System. In *IEEE Proc. of Real-Time Systems Symposium*, pages 257 – 266, December 1990.

[PDC92] T. Parr, H. Dietz, and W. Cohen. PCCTS Reference Manual, Version 1.0. *ACM SIGPLAN Notices*, 27(2):88–165, February 1992.

[SR91] J. A. Stankovic and K. Ramamritham. *Hard Real-Time Systems Tutorial.* IEEE Computer Society Press, 1991.

[X3.89] ANSI X3.159. *Programming Language C.* American National Standard Institute, 1989.