# Real-time Constraint Specification in Object-Oriented Languages

**Alexander P. Pons[1] and Moiez A. Tapia[2]**

## Abstract

The design and development of real-time systems is often a difficult and time-consuming task [2,3]. Real-time systems are systems where both the functional and the temporal behaviors of the system are essential. The system's correctness depends not only on the results of computations, but also on the time at which these results are produced. Building such systems has become increasingly complex due to the increased use of such systems in larger and more complex applications.

The rapid increase in the use and success of object-oriented (O-O) concepts in the realization of non-real-time applications motivates us to apply these concepts to real-time programs. Incorporating real-time domain concepts within object-oriented notions in a consistent fashion is the emphasis of this paper. We introduce the concepts of Temporal Abstract Classes and Virtual Temporal Constraints extending the inheritance mechanism to accommodate these real-time domain features. This significantly adapts the benefits of inheritance directly to real-time development, drawing from the contributions that inheritance renders to the reuse and consistency of software development.

## Introduction

Two of the major advantages of object-orientated methods compared to procedural methods are the concepts of abstraction through encapsulation and inheritance [4,17]. Incorporating timing constraint specifications within an object-oriented framework would provide an extension of these concepts into the real-time development domain. There currently exist various object-oriented languages for real-time programming. The techniques utilized in these languages vary on how timing specifications are incorporated into the language and linked to the object-oriented features.

In the literature there exist various languages [7,8] that provide programmers a means of expressing timing constraints within the program code. These constraints are used during feasibility and schedulability analysis and later conveyed to a real-time scheduler as directives to realize the real-time application. A number of requirements are defined for expressing real-time constraints. These requirements specify when a code block should start, when it should finish, whether it is a periodic or single invocation, how much processing time it may take, and how long the execution may take. The processing time is the difference between the start time of a code block and its completion, while the execution time is the length of time during which the code block is actually executing. Since part of the processing time may be spent in the execution of other code blocks, the processing time is at least as large as the execution time. Our time constraint mechanism is not completely unlike those found in other languages discussed in section 3.0. It differs in the location, usage and enforcement of the constraint specification to more readily support the object-oriented concepts.

## Object-Oriented Real-time Languages

Ideally, a real-time object-oriented language must provide sufficient real-time specifications as one may expect from a real-time language, while integrating these specifications within the object-oriented tapestry [5,6,11,13]. The significant aspects in object-oriented real-time modeling can be identified as:

1. The use of inheritance and redefinition of real-time constraints through the inheritance hierarchy and extension of the inheritance of the state and behavior of a class to include the definition of temporal constraints.

2. The reuse of the temporal constraint specifications through the inheritance mechanism and across class boundaries. These temporal constraints can be referenced to formulate new constraints in a consistent manner in classes.

3. Time-abstraction seems like a natural approach to specifying timing constraints, where the constraint is defined at the class definition and then associated with the class implementation. The temporal behavior of an object is made independent of the object's actual implementation by relating the temporal constraint to a labeled code block. This concept is similar to a method's signature that conveys the necessary behavioral information to use a method without considering its implementation. Consequently, a labeled code block can be given a temporal constraint using its signature without considering its specific association within the code, such as where the code block starts and ends a labeled code block comprises a class method or a sequence of statements identified with a label.

[1] Department of Computer and Information Systems, University of Miami, P.O. Box 248294, Coral Gables, FL 33124, apons@miami.edu
[2] Department of Electrical and Computer Eng., University of Miami, P.O. Box 248294, Coral Gables, FL 33124, mtapia@miami.edu

## Time Constraint in other Object-Oriented Languages

In most real-time languages, a timing construct is associated with a method by placing it in one of the three places: 1) at the method header declaration, 2) within the method's implementation as a code block, and 3) with the sending of a message (function call) or a statement in the method's implementation. The first is referred to, as being at the operation level since it is associated with the method specification, while the other two are considered statement level constraints. Timing constraints on statements are used to control execution depending on external behavior, while operation level constraints apply to a logical behavior. A logical behavior is a method, such as a calculate or process method which comprises the logic of the program. The following is a summary of how timing constraints are expressed in some significant real-time object-oriented languages.

**MPL [14]:** Timing constraints can only be expressed within methods; these timing constraints are imbedded in the implementation of the method and cannot be separately inherited nor referenced by derived classes.

**RTC++ [12]:** This permits timing constraints to be specified at a method's header or imbedded in the method body. Although expressing timing constraints at the method declaration level provides timing-abstraction, timing constraints cannot be separately defined/redefined nor inherited since they are solely associated with a method header.

**FLEX [9]:** Timing constraints may only be associated with statements. Labels can be attached to code blocks with constraints in order to refer to the start and finish time of the corresponding block. However, the constraints themselves cannot be referred, precluding their inheritance and redefinition.

**RealTimeTalk [16]:** Timing constraints are not permitted to be declared for derived classes; thus neither the inheritance mechanism nor constraint referencing is possible.

**DROL [15]:** Timing constraints for message sends may only be declared in method bodies. Timing constraints for message reception specify the worst-case time and may only be declared at the object interface. DROL allows, similar to RTC++, for statement and operation level specification.

These languages establish timing constraints using various styles; neither of them imposes the requirement or enforces class level timing constraint specification. The approach proposed in this paper will localize all timing constraints at the class level, which has the implications of utilizing inheritance, abstraction and defining virtual timing constraints.

## Class Description Model

These concepts will be expressed in a language independent manner utilizing Nelson's [10] class description. This simple and succinct manner of representing class definitions, without using any specific language, is possible with the following scheme:

**Class <class_name>**
**Superclass: <superclass1>, <superclass2>, . . .**
**ClassVar: <classvar1>, <classvar2>, . . .**
**InstanceVar: <instvar1>, <instvar2>, . . .**
**Method: <mthname1>, <mthname2>, . . .**

Superclass are the class's base classes if any, and ClassVar are any class-level variables. InstanceVar and Methods identify corresponding object-level variables and functions. The real-time specific features are described using this language independent class definition since they are language independent and can be applied to most object-oriented languages. Once an object-oriented language is extended with these real-time concepts, a preprocessor would need to be developed to translate a program in the extended language to a program in the unextended language for standard compilation.

## Timing Facilities

In the approach, all timing specifications are specified in the same location where functional descriptions are placed, this being the class definition. All objects instantiated from a class will share the same functional and temporal behavior. Elevation of temporal behavior to the class level similar to the functional behavior of classes will enable these temporal behaviors to acquire capabilities comparable to those that the functional behavior possesses. For example, if an object is needed with an identical functional behavior and different timing constraints, a new class can be derived, redefining the timing constraint while maintaining the other capabilities of the base class.

**Exploitation of inheritance anomaly concept:** When specifications in a base class implementation need to be modified in a derived class of the base class, the inheritance of the derived class in this case is said to have inheritance anomaly when the specification cannot be redefined separately from its application in the implementation forcing code duplication. Consider the effects of embedded temporal constraints within a method's implementation. This makes it impossible to redefine either the temporal constraint or the method implementation without redefining them both in a derived class. Inheritance anomaly is not inherent in the real-time domain but is a consequence of the language permitting the embedding of specifications (temporal, synchronization, semantics, etc).

The following Example 1 depicts how inheritance anomaly will require a duplication of the read function in the Sensor class and create another class, FastSensor, to use a faster read function, even though all that was necessary was to change the embedded timing constraint. This would be avoided if both the code and constraint could be inherited separately, thus both the code and the constraint being capable of redefinition independent of each other.

**Example 1**

```
Class Sensor
   Superclass:      None
   ClassVar:        None
   InstanceVar:     None
   Method:          int read(void)

   Method Implementation:
      int read(void)
         . . .
         within(20ms) { Constraint Statements }
         . . .
```

```
Class FastSensor
   Superclass:      Sensor
   ClassVar:        None
   InstanceVar:     None
   Method:          int read(void)

   Method Implementation:
      int read(void)
         . . .
         within(10ms) { Constraint Statements }
         . . .
```

A consequence of the use of inheritance anomaly is that the class implementation and temporal constraints are reusable apart from each other. This is achieved when the association of a temporal constraint is applied to a code block by means of a labeled sequence of statements. A sequence of statements with a temporal constraint will be designated with a label at the class definition level permitting separate reuse and redefinition. In extracting specific timing constraints in this fashion, the following outcomes result:

**Consistency of Specifications:** The definition of constraint specifications at the class definition provides for a consistent usage of these constraints. Such constraints could be referenced by other constraints used in various constrained blocks in the class it is defined or in derived classes during inheritance. Inheritance enables the reuse of verified code with the option of customizing the code.

**Redefinition of Specification:** The inheritance mechanism can be used to derive a class, which redefines both temporal or functional behavior of its base class without suffering from inheritance anomaly. The result is that timing constraints established by the base class can be redefined in the derived class. The benefits stem from not having to reimplement code blocks when their time constraints change or that a method in the base class can be redefined in the derived class while maintaining the base class time constraints.

**Virtual Temporal Constraints:** Virtual temporal constraints are defined as temporal constraints in a base class that are left unspecified and that must be specified fully in its derived classes. This allows a code block to be defined with timing constraints specified incompletely. The timing constraints, then, must be defined completely in its derived class(es). Failure to define temporal constraint specifications completely in derived class(es) will result in the generation of an error by the Real-time PreProcessor (RTPP)[3], preventing compilation of a real-time program until its temporal constraints are completely defined. This is similar to the concept of a virtual method, which corresponds to a behavior which is not implemented in a base class and which must be implemented in its derived classes. Classes that have at least one virtual temporal constraint are called Abstract Temporal Classes, a parity with Abstract Classes, which contain at least one pure virtual method.

**Temporal Abstraction:** Abstraction has been discussed as one of the main concepts that further the development of complex programs. Consider that functional-abstraction hides the actual implementation of a function or method, making available only the method's signature (name, return-type, parameter type and number). The programmer utilizes the method according to its signature, while ignoring the details of the method. Temporal abstraction accomplishes a similar objective in that from a programmer's perspective the timing constraint is a reference through a temporal specification name which permits the usage of the timing specification without the need for detailed information concerning the specification. The use of the specification is independent of how the timing constraints are composed so that changes made to the constraint do not affect its use. The name of the timing specification should be descriptive enough to make apparent its application, avoiding details in its association with code blocks.

[3] Once these features are added to an OOL, it becomes a new language, called Real-time Extension. A preprocessor called RTPP will translate the program in Real-time Extension to an original language so that it can be compiled.

## Code Blocks

Timing constraints can be applied only to statement sequences. A statement sequence is defined as a statement or sequence of statements that are associated with a name. A function, for example, is a type of code block that can be used to illustrate the application of a timing constraint. In order to impose timing constraint on a sequence of statements contained within a function, we require the notion of a labeled code block to be defined. A function is a type of labeled code block since the function name represents a sequence of statements; this concept is extended to associate a name at the statement-level.

**Labeled Code Block**: Applying a constraint on a code block requires the identification of this code block in a function or a method. This is accomplished by labeling the statement or a sequence of statements for which a constraint is to be applied. We shall use the following form to identify a constraint code block: **@Label: { constrainted-sequence-of-statements}@**

The **Label** should be descriptive enough to provide timing abstraction where the actual statements are hidden from the outside, similar to a method signature and its implementation. To establish a label **@ValMonitorDuration** for the sequence of statements in the block, see Example 2 below:

**Example 2**

```
void checkVal()
 // unconstrained statements
@ValMonitorDuration: {    //start block
   // sequence of constrained statements
}@                              // end block
// unconstrained statements
```

## Class Level Timing Constraints

In order to specify the timing of an object-oriented code, we have modified the class composition with a new section called the **constraint** section. Similar to the other sections, access controls can be applied to restrict the visibility of the characteristics of a class definition [1]. The constraint section can be used to apply various types of constraints on the class implementation. Since our concern is with real-time issues, the constraint section will be used to specify Temporal Specifications associated with statement sequences.

```
Class  <class_name>
       Superclass:        <superclass1>, <superclass2>, . . .
       ClassVar:          <classvar1>, <classvar2>, . . .
       InstanceVar:       <instvar1>, <instvar2>, . . .
       Method:            <mthname1>, <mthname2>, . . .
       Constraint:        <constraint1>, <constraint2>, . . .
```

## Temporal Specification

A **Temporal Specification** consists of a mandatory part, called **TimeSpecName,** and optional parts called **ConstraintBlockList** and **TimeConstraint.** We define the general form of a Temporal Specification as: **TimeSpecName[ConstraintBlockList: TimeConstraint]**

**TimeSpecName:** This is a unique name representing a temporal constraint. The name has a class scope. So it must be unique within a class definition. Its name should provide the abstraction of the intended timing constraint similar to a method's name. The TimeSpecName is used to reference attributes pertaining to a temporal constraint. For example, when defining a Temporal Specification in terms of another Temporal Specification, the TimeSpecName will be used to refer to its TimeConstraint. The TimeSpecName can be qualified to refer to Temporal Specifications in classes other than the class defining a Temporal Specification. This is accomplished by qualifying the TimeSpecName with the name of the defining class. During the inheritance of Temporal Specifications, the TimeSpecName is inherited in the derived classes and is used to access and/or redefine the parts of the Temporal Specification. This will be further elaborated in the Inheritance of Temporal Specification in a later section.

**ConstraintBlockList:** This is a list of names of methods and/or code blocks to each one of which a timing constraint is to be applied. The name of a code block is the same as the name of the label associated with the block in the program. A name in the list may include the special characters, * and %. This enables it to represent names that could be derived from it according to the following rules: (1) The * represents zero or more wildcard characters, while the % represents one character wildcard. (2) These special wildcard characters can be placed in any location within a name, except for the first character in the name. (3) The % wildcard can be used more than once in a name. (4) The * wildcard can be used only once in a name.

**TimeConstraint:** This is used to establish a temporal constraint. It consists of a TimeConstraint expression utilizing other Temporal Specifications (their TimeSpecName), integer constants and basic arithmetic operator (+,-,*,/) in formulating the current timing constraint specification. The expression must evaluate to a time-constant, which will become a parameter to a time specification.

The TimeConstraint provides the **within(time-constant)**, **at(time-constant)** and **before(time-constant)** expressions. **within** specifies the duration of execution. This can be referred to as an interval constraint such as a deadline for a sequence of statements. **at** and **before** specify event time of execution. They express the constraint of starting time and ending time, respectively for a sequence of statements within a labeled code block, or a referenced code block or the start of the program. The **cycle(***start-time; end-time; period; deadline)* can be applied to class functions. The cycle is for indicating a periodic task with the start-time, end-time, period and deadline, all of these parameters being time-constants.

## Timing Constraint Specification

Example 3 utilizes class level timing specifications to establish temporal constraints consistent with object-oriented concepts:

**Example 3**

```
Class Sensor
   Superclass:     None
   ClassVar:       None
   InstanceVar:    None

   Method:
      int checkRange( ),
      int validSensorVal( ),
      int read(char* data, int size) ,
      int write(char* data, int size),
      int sensorGet( ),
      int sensorClear( ),
      int getTriggerVal( ),
      int slowRead( ),
      int calcRPM( )

Constraint:
   ExternalAccess[read, write: within(20ms)],
   SensorCheck[sensor*: within(10ms)],
   SlowSensorCheck[slow*:within(2*SensorCheck)]

Method Implementation:
int Sensor::getTriggerVal(void)
   . . .
   @sensorLoop:{
        . . .
    }@
   . . .
```

This example uses the **ExternalAccess[read, write: within(20ms)]** Temporal Specification to indicate that the **read** and **write** functions have to terminate within 20 milliseconds. The following Temporal Specification **SensorCheck[sensor*: within(10ms)]** indicates that any name in the ConstraintBlockList which begins with the **sensor** prefix must complete within 10 milliseconds. This Temporal Specification will be applied to the **sensorGet** and **sensorClear** function since they match the specified pattern. Also, in the function **getTriggerVal,** there is a sequence of statements inside the braces of the labeled code block called **sensorLoop,** which must complete within 10 milliseconds. This constraint is applied based on the prefix **sensor**.

When referencing Temporal Specifications to assign values to related Temporal Specifications, the time-constant of the referenced Temporal Specification is the part that is accessible, not any of the other composing parts of the Temporal Specification. In the previous example, the TimeConstraint of the Specification **SlowSensorCheck** is determined using the time-constant in the TimeConstraint of the Specification **SensorCheck** which is 10 milliseconds rendering a time-constant of 20 milliseconds.

## Temporal Abstract Classes and Virtual Temporal Specifications

The inheritance mechanism used for Temporal Specifications is similar to that normally defined for class attributes. This includes the specification of timing constraints at a base class that can be redefined in its derived classes. When a constraint is not fully specified at the base class, then this class is referred to as a **Temporal Abstract Class**, which indicates that at least one of its Temporal Specification is a **Virtual Temporal Specification**. In the approach proposed in this paper a Virtual Temporal Specification has been composed where the time-constant in the TimeConstraint section has been set to **0**. A Virtual Temporal Specification associates a list of names of a ConstraintBlockList with a TimeConstraint whose TimeNotation must be specified by a derived class prior to the program being analyzable.

Let us extend the class in Example 3 with inheritance and Virtual Temporal Specification as shown in Example 4. The **CalculateSensorValue** Temporal Specification defined in the **Sensor** class is a Virtual Temporal Specification since it meets the established criteria. Since the Sensor class has at least one Virtual Temporal Specification, it qualifies the class as a Temporal Abstract Class. This categorization of the class will preclude its usage for object instantiation. The derived class **LengthSensor** given below will have to define the Temporal Specification **CalculateSensorValue** before the timing specifications can become complete.

**Example 4**

**Class Sensor**
**Superclass: None**
**ClassVar: None**
**InstanceVar: None**
**Method:**
**int checkRange( ),**
**int validSensorVal( ),**
**int read(char* data, int size) ,**
**int write(char* data, int size),**
**int sensorGet( ),**
**int sensorClear( ),**
**int getTriggerVal( ),**
**int slowRead( ),**
**int calcRPM()**

**Constraint:**
**ExternalAccess[read, write: within(20ms)],**
**SensorCheck[sensor*: within(10ms)],**
**SlowSensorCheck[slow*:within(2*SensorCheck)],**
**CalculateSensorValue[calc*: within(0)]**

**Class LengthSensor**
**Superclass: Sensor**
**ClassVar: None**
**InstanceVar: None**
**Method:**
**int read(char* data, int size),**
**int calcSpeed( )**

**Constraint:**
**FastAccess[read:within(5ms)],**
**CalculateSensorValue[calc*:within(25ms)]**

In the class Sensor, the Temporal Specification **CalculateSensorValue[calc*: within(0)];** is a Virtual Temporal Specification. This temporal specification matches the method calcRPM, associating it with a virtual TimeConstraint. The derived class LengthSensor redefines the Temporal Specification, providing a time-constant of **25** milliseconds, which now establishes a complete TimeConstraint for all labeled code blocks beginning with the prefix **calc.**

Inherited Temporal Specifications will be applied to any code block labels not already matched in the derived class, if their names are associated with the ConstraintBlockList part of these Temporal Constraints. Once all of a class's Temporal Specifications have been utilized to associate TimeConstraints to code blocks, then inherited Temporal Specifications are used to continue the process. A code block previously assigned a TimeConstraint will not be considered for assigning a TimeConstraint even though its name matches a ConstraintBlockList pattern of the inherited Temporal Specifications.

For example, the calcSpeed method of the LengthSensor class is assigned a temporal constraint from the local definition of the CalculateSenssorValue. This same temporal constraint will be applied to the inherited calRPM method of the Sensor class since it will redefine the inherited CalculateSenssorValue to 25 milliseconds. The read function of LengthSensor will be applied the FastAccess temporal specification of the class LengthSensor. When the inherited temporal specifications are considered in associating temporal constraints to the labeled code blocks of the LengthSensor class. The ExternalAccess temporal specification establishes a temporal constraint for the method read, but since LengthSensor has already been associated with a temporal constraint specified in the LengthSensor class, it will not affect the already assigned temporal constraint. Hence, this provides the redefinition of temporal constraints for methods in the derived class affecting derived class methods and methods inherited from the base class that match the ConstrainedBlockList names of the derived class's temporal specifications.

## Behavior of Temporal Specifications

The following example demonstrates the association of ConstraintBlockLists and TimeConstraint in a class and through inheritance. The notation used throughout these examples uses T? and M?, where ? is some integer number to indicate a Temporal Specification and the name of a constrained code block, respectively. Observe the following convention `Ti<-TCj` depicts the association of a TimeConstraint TCj with a Temporal Specification Name Ti, and `Mp<-Tq` depicts the association of a Temporal Specification name Tq with a code block name Mp. When ambiguity arises, qualify the name with a class name using the :: symbol. The subscripts i, j, q and p are integers used to develop names.

### Within a class, no inheritance:

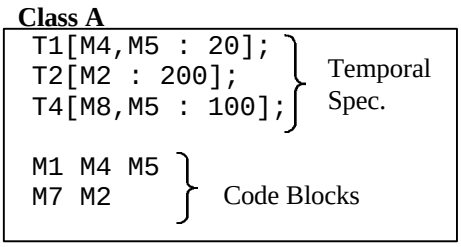
1. There cannot exist more than one Temporal Specification with the same name; this will generate an ambiguity error.

2. Each Temporal Specification is applied in the order they are placed within the class. This gives precedence to the Temporal Specifications found first within the class. Each name in every ConstraintBlockList is applied to as many labeled code blocks and method names that match the name pattern. Once a code block is given a TimeConstraint, it will not be redefined if its name matches another Temporal Specification. If a code block name finds a match after its first match, it will generate a warning

message during the temporal semantic analysis phase. For example, consider the following two Temporal Specifications in a class: **T1[calcSpeed : within(2ms)];** and **T2[calc* : within(1ms)];** A code block with the name calcSpeed will match both of these Temporal Specifications, but T1 will be applied since it appears before T2 in the class definition. Also, a warning will be generated when T2 matches calcSpeed during the scanning process for this purpose.

3. If there are ConstraintBlockLists that do not match any of the class's code block names, then this will generate a warning message stating that there are unused ConstraintBlockLists in the respective Temporal Specifications.

**Example 5**




```
T1 <- 20    M4, M5 <- T1
T4 <- 100   M2     <- T2
T2 <- 200
```

- M4 and M5 match T1 and get 20.
- M2 matches T2 and gets 200.
- M8 in T4 generates a warning since it is specified and does not match any of the class's code blocks.
- M5 is given another Timing Constraint in T4, which is ignored as it already has been given a constraint with T1 encountered first. This generates a warning message, indicating an attempt to redefine the constraint of M5.
- M7 is a code block that does not match any name in the ConstraintBlockList for any of the Temporal Specifications.

## Within a class, with inheritance:

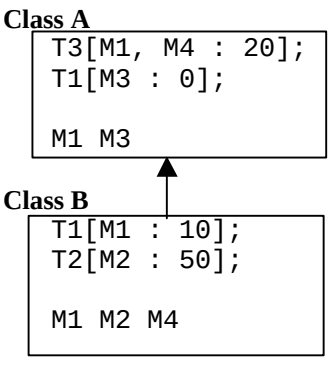
1. Using the ConstraintBlockList, denoted by Mi, of a derived class, i being an integer, attempt to match the pattern with local code blocks.

2. Once done with the local level, search in the base classes from which the derived class inherits, for code block names which match the pattern to apply this constraint. This gives precedence to derived class Temporal Specifications over inherited specifications from the base class(es).

3. Once the derived Temporal Specifications are exhausted, start with the first inherited base class Temporal Specifications and commence searching for code block matches at the derived class and then up the hierarchy of classes.

4. Applying a Virtual Temporal Specification to a code block will result in an error being generated indicating that the temporal characteristics of the program are incomplete.

**Example 6**




```
B::T1 <- 0  B::M1 <- B::T1
B::T2 <- 50 B::M2 <- B::T2
A::T3 <- 20 A::M3 <- B::T1
A::T1 <- 200B::M4 <- A::T3
```

- M3 is inherited and M1 is redefined in Class B.
- The Temporal Specifications T1 and T2 are associated with Class B's M1 and M2.
- The inherited Temporal Specification T3 from Class A will establish a TimeConstraint on M4 but not M1 since it has already been given a TimeConstraint.
- The Temporal Specification T1 from Class A is a Virtual Temporal Specification. When it is redefined in Class B, then M3 is applied the TimeConstraint for the T1 Temporal Specification of Class B.
- Class A is an Abstract Temporal Class, requiring a redefinition of T1 in the derived class. M3 will be assigned the TimeConstraint of Class B Temporal Specification T1 despite its association with a Virtual Temporal Specification as T1 is redefined from its Class A location.

## Conclusion

Realization of the benefits of object-oriented concepts within the real-time domain requires an extension of OO concepts. These extensions must allow application code to be reused separately from its real-time specifications. This promotes the reuse of both application code and timing specifications. When separation is not possible, changes made to the application requirements in sub-classes may result in excessive redefinition. Real-time programs are, in general, difficult to design and verify. The inheritance mechanism can be useful in reusing well-defined and verified real-time programs. The definition of Abstract Temporal Classes and Virtual Temporal Constraint Specifications extend the object-oriented methodology to the real-time domain in a consistent manner.

A new language, Hard Real-Time C++, has been developed as an extension of C++. A Real-Time preprocessor has been designed and developed to the real-time extensions to the C++ language and provide a temporal semantic phase. The temporal semantic phase assimilates the real-time extensions while identifying errors and warnings meaningful within the context of the real-time program. The preprocessor extracts these annotations and keywords from the real-time program producing a C++ program for normal compilation. It also generates a file, which records the various time-constrained code blocks and their respective TimeConstraints. This file serves as input to static analyzing tools identified along with the preprocessor to conduct feasibility and schedulability analysis. The preprocessor and feasibility tool which complement the extension of an object-oriented language to the real-time domain provide real-time developers with much needed assistance in the design and implementation of real-time programs.

## References


1. Pons, Alexander, (1998) “An Object-Oriented Language for Hard Real-time Systems”, Ph.D. Dissert., Univ. of Miami, FL.

2. Halang, Wolfgang., (1992) “Real-time Systems: Another Perspective”, The Journal of Systems and Software, Apr. 101-108.

3. Stankovic, J. and Ramamritham, K., (1998) “Hard-Real-time Systems”, IEEE Tutorial, IEEE Press, Washington DC.

4. Booch, Grady., (1986) “Object-Oriented Development”, IEEE Transactions on Software Engineering, Feb. 211-221.

5. Gligor, V. and Luckenbaugh, G., (1983) “An Assessment of the Real-time Requirements for Programming Environments and Languages”, Proceedings of the IEEE Real-time System Symposium, Washington DC., 3-19.

6. Kligerman, E., and Stoyenko, A., (1986) “Real-time Euclid: A Language for Reliable Real-time Systems”, IEEE Trans. on Software Engineering, Vol. SE-12, No. 9, Sept.

7. Stoyenko, A., (1992) “The Evolution and State-of-the-Art of Real-time Languages”, The Journal of Systems and Software, Apr. 61-84.

8. Halang, W. and Stoyenko, A., (1990) “Comparative Evaluation of High-Level Real-time Programming Languages”, Int. Jour. Time-Critical Comp. System, Vol. 2, 365-382.

9. Lin, K. and Natarajan, S., (1988) "Expressing and Maintaining Timing Constraints in FLEX", IEEE Proc., 96-105.

10. Nelson, M., (1990) “Object-Oriented Real-time Computing”, Naval Postgraduate, Monterey, CA, Tech. Report No. NPS52-90-025.

11. Pereira, C., (1993) “Putting OO to work: Results from Applying the Object-Oriented Paradigm during the Development of Real-time Applications”, Fifth Euromicro Workshop on Real-time Systems Proceedings. Apr. 166-170.

12. Ishikawa, Y., Tokuda, H., and Mercer, C., (1992)“An Object-Oriented Real-time Programming Language”, IEEE Comp., Oct. 66-73.

13. Bihari, T., Gopinath, P., and Schwan, K., (1989)”Object-Oriented Design of Real-time Software”, IEEE Software Trans., 194-201.

14. Nirkhe, V., Tripathi, K., and Arrawala, A., (1990)“Language Support for the MARUTI Real-time System”, IEEE Soft. Jour, 257-266.

15. Takashio, K. and Tokoro, M., (1992) “DROL: An Object-Oriented Programming Language for Distributed Real-time Systems”, Proceedings of OOPSLA’92, ACM Press, 276-294.

16. Brorsson, B., Eriksson, C. and Gustafsson, J., (1992) “RealTimeTalk: An Object-Oriented Language for Hard Real-time Systems”, Proceedings of IFAC International Workshop on Real-time Programming.

17. Cox, J. B., (1986) “Object-Oriented Programming: An Evolutionary Approach”, Addison-Wesley.