Programming Language Concepts for Multimedia Application Development

Oliver Lampl, Elmar Stellnberger, and László Böszörményi

oliver.lampl@hollomey.com, estellnb@yahoo.de,
laszlo.boeszoermenyi@itec.uni-klu.ac.at

Abstract. Multimedia application development requires features and concepts currently not supported by common systems programming languages. This paper introduces two new minimal language extensions increasing expressive power, safety and optimization possibilities in multimedia programming. New loop statements are presented to shorten multidimensional array access and optimize its execution. Furthermore, a new data type concept is presented to allow quality of service (QoS) definition on data type declaration level. Both have been implemented in Modula-3 and C#.

1 Introduction

Substantial parts of programs processing multimedia data follow some very common patterns:

1. In the compression/decompression/transformation part of such programs, large multidimensional numerical arrays are partitioned into small independent blocks and processed by algorithms, like the Discrete Cosine Transformation (DCT).
2. In the video streaming and play-back part, long sequences of data (e.g. video frames) are processed and/or transmitted periodically, under so-called "soft real-time" constraints.

The manually created code for these recurring patterns is typically cumbersome, error-prone and inefficient. These observations suggest that we could give good support for multimedia programming on the level of a programming language. A vast number of multimedia query languages resp. language extensions exist [15], nevertheless, to our knowledge, no language support for multimedia systems-programming exists. We argue in our paper that such a support is advantageous and easily possible.

The first pattern obviously calls for a simple, automatic parallelization. To handle the second pattern, we need a notion of time, and a way to express Quality of Service (QoS) constraints. Instead of defining a brand new language, we investigated the possibilities of extending some existing programming languages with the minimal necessary features, considering the following basic principles: Add an extension to a language only if the following conditions are fulfilled:

1. The new feature enhances the expressive power of the language considerably.
2. The safety of programs using the new feature is enhanced.
3. The new feature enables some automatic optimizations.

Everything else should rather be put into a library than applied as a language extension. Under these premises we suggest the following extensions for general purpose programming languages:

1. A *foreach* and a *forall* statement enabling compact and safe expression, and automatic parallelization of typical video transformation code, operating on independent blocks of data.
2. A time dimension, which can be added to any existing scalar or array type as an n+first dimension.
3. A very simple first-order logic based language extension, enabling to express QoS constraints.

The actual language extensions were designed both in Pascal- and C-style and were implemented in two well-known representatives of these language families: Modula-3 [13] resp. C# [11]. The parallel development helped us a lot to separate the essence of a new construct from the syntactic sugar and it was - by the way - the source of a lot of fun.

2 Related Work

2.1 Parallelism

Generally, two different approaches exist to introduce parallelism. In the synchronous approach one instruction is used to work on multiple data elements. The asynchronous approach on the other hand allows the execution of different instruction streams simultaneously.

In [3] Philipsen and Tichy implement a machine independent *forall* loop both in a synchronous and asynchronous version targeting multiple architectures with shared and distributed memory. Furthermore they showed in [4] that with the use of an adequate working environment debugging of parallel programs written with *forall* loops is feasible and does not pose a big problem.

In [2] Knudsen introduces a queue to distribute the workload of an asynchronous *forall* loop on multiple execution units of a shared memory system as provided by multi processor and multi core computers. Implementing dynamic workload generation via distributing nested procedures by a queue causes very little overhead in the range of a few percents. This approach reaches a high utilization even in the last loop iterations without the need of static analysis.

We have adopted his approach both in our Modula-3 and C# implementation. In the latter, however, we pass objects instead of using nested procedures because nested procedures are not supported in C#. Supplying an asynchronous *forall* seems to be sufficient in most cases of multimedia programming. Asynchronous *foralls* are also open to vectorization thus allowing a further possibility for speedups.

In [14] Zima presents how dependence analysis can be used by compilers to automatize vectorization and parallelization. Source to source transformation using *forall* loops as a target construct is suggested. Explicit synchronization barriers are used rather than implicitly inserted barriers at the end of each parallel loop. The described techniques can also be used to verify the mutual independence of different loop iterations, a condition which is demanded but not yet checked by Knudsens approach, nor by ours. Loop carried dependencies would require explicit assumptions about loop behavior.

2.2 Quality of Service

Quality of service (QoS) is another important aspect of multimedia applications. A lot of different notations and specifications can be found to express QoS-related mechanisms like resource reservation, admission control, and adaptation. Jin and Nahrstedt provide a classification over existing QoS specification languages [9]. These specification languages try to cover most aspects of QoS and are defined on application-, user-, or resource-level to allow a user-friendly notation.

To apply QoS constraints directly at the programming language level not all aspects of QoS have to be met. When applying QoS on displaying of a video the most important QoS constraints are frame rate, delay, and jitter. These timing limitations can be expressed using temporal logic. In [1] Blair and Stefani introduce the first order logic based language QL to define and formally analyze QoS constraints. QL is based on an event model basically identified by three components: event types (1) , events (2) and histories (3).

1. Event types represent a particular state transition in a system (e.g. the arrival of a frame of video).
2. An event is an occurrence of an event type (e.g. the arrival of a particular frame of video).
3. The history represents a discrete sequence of events of the same event type.

To reflect a special occurrence of an event in the history the function $\tau(\varepsilon, n)$ is used, where n represents the nth occurrence and ε the event type. By applying this model, we can express a wide range of quality of service constraints. E.g. the throughput of video can be expressed (ε_r stays for frame reception):

$$\forall n, \tau(\varepsilon_r, n+k) - \tau(\varepsilon_r, n) \leq \delta$$

To be precise, this formula specifies that for all video frames, the difference in time between the arrival of the frame n + k and the frame n is less than a given value δ. The next example shows the definition of bounded execution time (ε_e stays for the emission, ε_r for the arrival of a frame):

$$\forall n, \left|\tau(\varepsilon_{e,n}) - \tau(\varepsilon_r, n)\right| \leq \delta$$

In this case, the maximum allowed delay between two different event types is specified for all of their occurrences.

In [1] Esterel is used for QoS monitoring. It is an imperative language specifically developed in order to assert the QoS compliance of networked applications. An

Esterel program consists of a set of parallel processes which execute synchronously and communicate with each other. Apart from its fancy signaling concept Esterel is quite minimalistic. Programmers may prefer a better integrated approach that is easy in practical deployment and that allows them to make use of their existing knowledge. However, we chose a different approach, see *QoS Monitoring*.

Our work concerning QoS is based on the event model of QL. It can describe most of the QoS constraints required in multimedia applications. Nevertheless it is limited and some constraints cannot be expressed like general reliability requirements such as Mean Time Between Failure or Mean Time To Repair.

3 Parallelism and Loops

3.1 Extended Loop Statement

Pixel manipulations can be implemented using multidimensional arrays. A lot of encoder or decoder implementations make use of such data structures. In programming languages like Modula or C/C++, such arrays are iterated using simple *for* statements. In Java and C# new loop statements have been developed in order to iterate over collections or arrays.

In C# the *foreach* statement [11] is used to iterate over expressions that can be evaluated to a type that implements the *IEnumerable* interface, or a type that declares a *GetEnumerator* method which then returns an object of type *IEnumerator* [8].

```
foreach ( type identifier in expression )
         embedded-statement
```

These enumerators iterate over the stored elements. For each element the embedded statement is executed.

In multimedia applications we often want to refer to the index of the elements accessed. Therefore we extended the *foreach* statement to define the expression for retrieving the elements of the array. This enables the programmer to define index variables which can be accessed during the loop. To avoid unpredictable side effects index access is read only.

```
foreach ( type identifier = expression in expression )
         embedded-statement
```

The following example represents a simple implementation of the discrete cosine transformation (DCT) as used for JPEG implementations [10] implemented in C#.

```
double value = 0;

for (int u = 0; u < 8; u++) {
 for (int v = 0; v < 8; v++) {

  for (int i = 0; i < 8; i++) {
   for (int j = 0; j < 8; j++) {
        value += (source[i,j] - 128)
               * Math.Cos(((2*i + 1) * u * Math.PI) / 16)
               * Math.Cos(((2*j + 1) * v * Math.PI) / 16);

     }
    }

   coefficients[u,v] = value / 4;

  }
 }
```

Instead of using four conventional nested *for* statements the code fragment can be reimplemented by applying two extended *foreach* loops without considering the size of the array being iterated over.

```
   foreach (double c = coefficients[u,v]
              in source) {

    foreach (int p = source[i,j] in source) {

     value += (p - 128)
               * Math.Cos(((2*i + 1) * u * Math.PI) / 16)
               * Math.Cos(((2*j + 1) * v * Math.PI) / 16);

    }

    coefficients[u,v] = value / 4;

   }
```

The extended *foreach* has been implemented in Modula-3 too.

```
FOREACH c = coefficients[u,v] IN coefficients VIA u,v DO

 FOREACH p = source[i,j] IN source VIA i,j DO

    value := value + ( FLOAT(p-128,LONGREAL)
           * cos(FLOAT((2*i+1)*u,LONGREAL)*Pi/16.0D0)
           * cos(FLOAT((2*j+1)*v,LONGREAL)*Pi/16.0D0));

 END

 coefficients[u,v] := value / 4.0D0;

END
```

The extended *foreach* loop expresses more clearly how elements are assigned within the loop. Furthermore, no array ranges have to be considered. The compiler internally generates the correct code for iteration and therefore provides more safety. No infinite loops can be created and the statement enables the programmer to express index based calculations with arrays. Like the original *foreach* in C#, the current element of the loop can be directly accessed.

3.2 Parallel Loop Execution

To optimize performance, parallelism can be added to the loop to distribute its execution into several threads. This executional optimization can be reached due to the fact that a *foreach* statement as defined in [11] does not necessarily guarantee a special order of execution. The semantic only defines that each element of the given collection or array is accessed. This fact can be used to implement a *foreach* loop in which each of the iterations can be executed simultaneously. The syntax is very similar to the shown *foreach* or the extended *foreach* statement, but the semantics differ.

```
forall ( type identifier in expression )
         embedded-statement
forall ( type identifier = expression in expression )
         embedded-statement
```

The block executed each time the loop iterates is embedded into a *job class*. The instance of this class represents a job which is executed for each loop iteration. Instead of executing the jobs in sequential order, they are put into a queue to feed workers which can operate in parallel. These workers are controlled by a management framework which observes the execution of all workers. Furthermore, it ensures that sequential execution follows after all parallel work is done. The compiler itself generates code to fork the workers, distribute the work to each worker and synchronize all workers when all work has been completed. After all parallel work is done, the program continues normally.

3.3 Parallel Processing Framework and Results

The default worker pool implementation is integrated into the system class library of Mono [7]. To give programmers the ability to implement their own worker pools the default behavior of jobs and the worker pool is defined by the interfaces *IJob* and *IworkerPool* (see figure 1). The worker pool can be exchanged during runtime by replacing the default worker pool implementation with the *WorkerPoolFactory*.

Each time a *forall* statements loop body is executed, an *IJob* object is created and added to the worker pool. At the end of the loop the *waitTillFinished* method is called to ensure that work has been done before continuing. The implementation uses the default multithreading libraries. The same behavior can be achieved using asynchronous method invocation. This feature of the Common Language Runtime [12] can be used to inherently introduce concurrency into a program. The *forall* statement has also been implemented in Modula-3 using nested procedures [5, 6, 16] instead of passing objects.

But parallelism is not without issues. The programmer has to consider the overhead for thread creation, control and the cost of object initialization. Normal loops can be terminated using *return* or *break* statements. This cannot be easily achieved when using parallelized execution. So the use of these statements has been forbidden, because of the misleading semantics. If *break* or *return* is used, the programmer wants the execution of the loop to stop, but the parallel execution disables immediate loop termination. Another problem is exception handling within the loop. This is achieved by catching exceptions within the worker threads, and throw them at the end of the execution to allow the programmer to use the default exception handling mechanisms of the language.

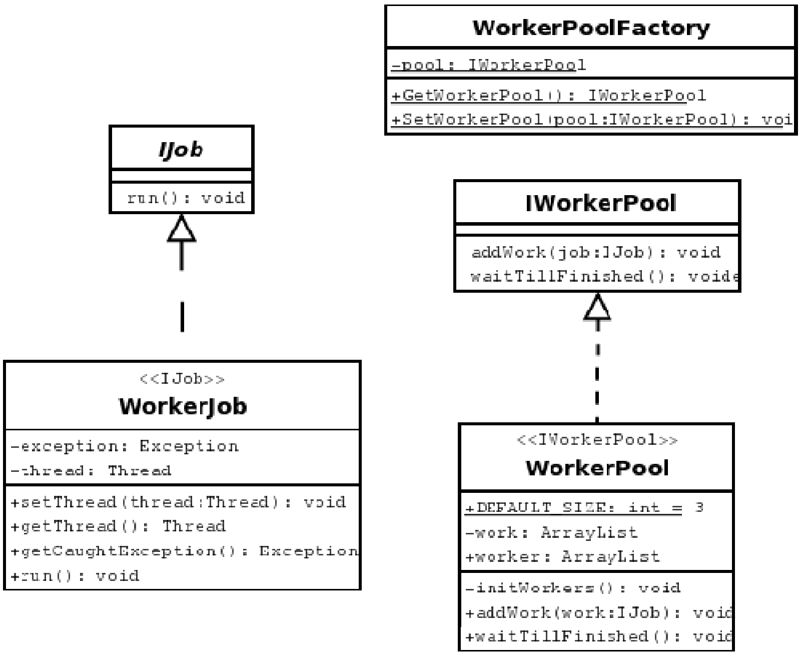


Fig. 1. Classes involved in the parallel loop statement

The performance enhancement is demonstrated using the simple block based DCT (see prior example). The *forall* statement is applied to parallelize the processing of blocks where the number of threads and the number of blocks are varied. The measurement was done on a quad processor machine showing linear scalability and little overhead (see figure 2 and 3).

The execution time of the *forall* statement using one single thread is only a few percents higher than the execution time of an implementation using *for* statements. This overhead is caused by object initialization and thread control. When increasing the number of threads, the execution time decreases significantly, but when the number of threads reaches the number of processors, the management overhead grows considerably. Similar results are presented in [2] which proofs the efficiency of our implementation.

At this point the presented concept will be evaluated according to the criteria defined at the beginning of this paper:

1. The expressive power of the extended *foreach* statement has been demonstrated by rewriting a DCT. The new statement tries to express the array loops with more simplicity in syntax but of course higher complexity in semantics.
2. The safety of the programs is enhanced due to the fact that the loop termination condition is hidden in the compiler generated code and guarantees that the loop comes to an end after it has accessed all elements of the data structure.
3. The *forall* loop can be used to optimize the execution time of the program (demonstrated by examples and graphs), where the loop body is separated into work packages and put into an efficient parallel processing framework.

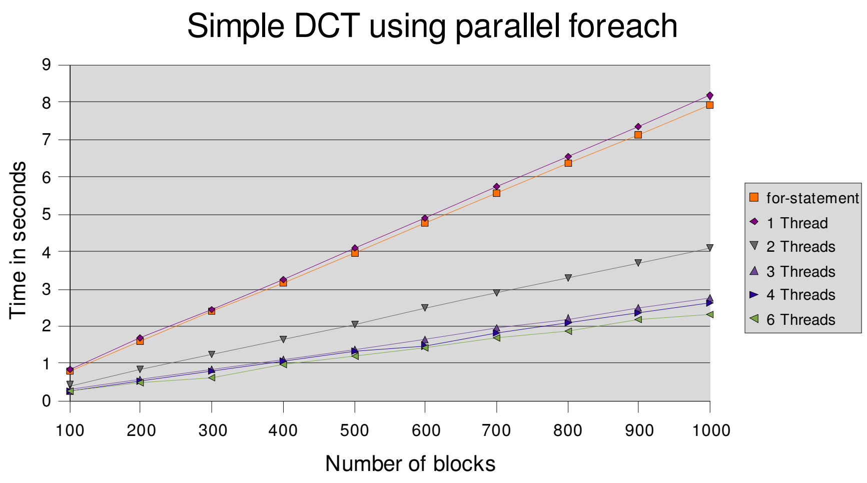


Fig. 2. The amount of time needed to process a given number of blocks using a set of threads

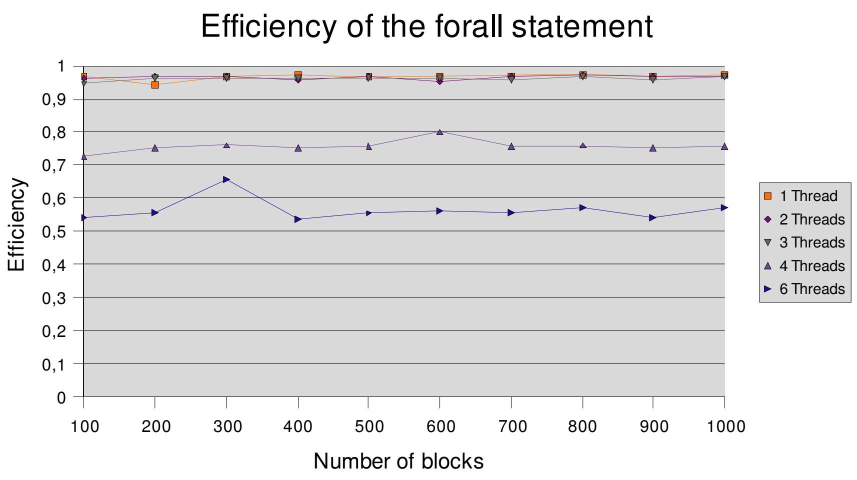


Fig. 3. The efficiency of the distribution to multiple workers

4 Monitoring QoS Constraints

4.1 Quality Aware Data Types

Instead of implementing QoS monitoring with Esterel as shown in [1], we introduce the concept of quality aware data types. The declaration of said types allows the programmer to specify the quality of service contract for a simple data type. A special assignment operation is used to examine declared constraints and cause exceptions in case of constraint violations.

The intent of quality aware data types is not to enforce a specified constraint. To achieve this, real time systems have to be used. In case of multimedia applications users might agree to quality of service changes if the cost is reduced, so we just have to check possible violations and inform about their occurrence.

Quality aware data types are declared by an additional dimension of time. This n+first dimension is declared using the token [~].

```
type [~] identifier
```

The dimension of time can be parameterized to specify quality of service constraints. The initialization of the type can be expressed in a static way for primitive data types with a constant constraint or using the *new* command to allow dynamic creation. The dynamic way provides exchangeability of the quality of service object to implement adaptive quality of service features.

```
type[~(IQoSObject)Identifier]

type[~] identifier = new type
                            [~(IqoSObject)identifier]
```

The quality of service parameter definition is encapsulated within an *IQoSObject*. This interface is used to implement QL like quality of service constraints, which are controlled automatically by compiler generated code. The programmer may use available implementations of QoS constraints and can also add own code.

4.2 Implementation Issues

Quality aware data types are implemented using event histories. Such events can be evaluated by an *IQoSObject* which then decides whether the given constraint can be held or not. The event happens at the assignment statement and is recorded in the history. In order to distinguish between an assignment and a QoS monitored assignment we define the timed assignment operation "~=".

Each time a quality aware data type is assigned using the timed assignment operation, its currently embedded constraints are checked. If the check fails a *QoSException* is thrown, which is used to react upon constraint violation. This allows easy implementation of adaptive quality of service constraints, e.g. the frame rate can be changed or the size of frames can be reduced. The definition of an additional assignment statement is advantageous because it allows the programmer to decide if the quality aware data type is monitored for the current assignment or not. This can be compared to video processing. If we playback a video, we monitor QoS constraints to achieve correct frame rate, jitter and delay. In case of management operations on videos, e.g. format conversion or video analysis for meta data retrieval, we do not necessarily need to monitor QoS.

The first example demonstrates a quality aware numerical type which is assigned within a *for* statement to values of a given array. The quality aware variable *value* is defined with the constraint *QoSDelay*. This object is initialized with the variable to be monitored and a numerical value expressing the delay in milliseconds to slow down execution. Each time *value* is assigned using the timed assignment statement the execution is delayed to print one value per second to the console. The numerical value is streamed.

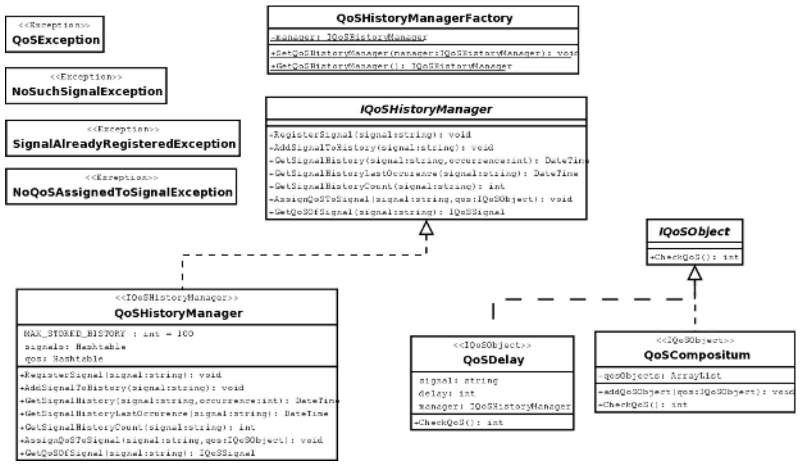
```
int[~new QoSDelay("value", 1000)] value;
int[] data = {1,2,3,4,5,6,7,8,9};
for (int i = 0; i < data.Length; i++) {
         value ~= data[i];
         Console.WriteLine(value);
}
```

The control structure for video transcoding applications is implemented using quality aware data types. The variable *output* is declared as a quality aware data type and initialized with th QoS constraint *QoSThroughPut*. The QoS constraint is used to monitor the throughput of frames. It is initialized to monitor that every 1000 milliseconds 25 frames are processed. The short code for transcoding ensures that the quality of service constraint for display is held if not, an exception is thrown to terminate the execution of the loop. If the programmer implements the quality of service check manually, the code would be much more complicated. The quality aware variable *output* limits its execution time while being assigned using the timed assignment statement.

```
public void transcode(FrameIterator frames) {
 IQoSObject constraint =
           new QoSThroughput("output", 1000, 25);
 Frame[~] output = new Frame[~constraint];
 try {
  foreach(Frame f in frames) {
   output ~= transcodeFrame(f); // transcoding function
   display(output); // display output
  }
 } catch (QoSException e) {
  Console.WriteLine("QoS Constraint Violation!");
 }
}
```

4.3 QoS Management Framework

The history of the events are recorded with the implementation of the *IQoSHistoryManager* (see figure 4). This class registers events, assigns quality of service constraints to these events, and stores their histories within a time frame. The default implementation available in the system class library is used by the compiler by default. To enable the programmer to provide its own implementation, the default history manager can be exchanged using the *QoSHistoryManagerFactory*.



<<Exception>>
QoSException
<<Exception>>
NoSuchSignalException
<<Exception>>
SignalAlreadyRegisteredException
<<Exception>>
NoQoSAssignedToSignalException
QoSHistoryManagerFactory
-manager: IQoSHistoryManager
+SetQoSHistoryManager(manager:IQoSHistoryManager): void
+GetQoSHistoryManager(): IQoSHistoryManager
IQoSHistoryManager
+RegisterSignal(signal:string): void
+AddSignalToHistory(signal:string): void
+GetSignalHistory(signal:string,occurrence:int): DateTime
+GetSignalHistoryLastOccurence(signal:string): DateTime
+GetSignalHistoryCount(signal:string): int
+AssignQoSToSignal(signal:string,qos:IQoSObject): void
+GetQoSOfSignal(signal:string): IQoSSignal
<<IQoSHistoryManager>>
QoSHistoryManager
MAX_STORED_HISTORY : int = 100
signals: Hashtable
qos: Hashtable
+RegisterSignal(signal:string): void
+AddSignalToHistory(signal:string): void
+GetSignalHistory(signal:string,occurrence:int): DateTime
+GetSignalHistoryLastOccurence(signal:string): DateTime
+GetSignalHistoryCount(signal:string): int
+AssignQoSToSignal(signal:string,qos:IQoSObject): void
+GetQoSOfSignal(signal:string): IQoSSignal
IQoSObject
+CheckQoS(): int
<<IQoSObject>>
QoSDelay
signal: string
delay: int
manager: IQoSHistoryManager
+CheckQoS(): int
<<IQoSObject>>
QoSCompositum
-qosObjects: ArrayList
+addQoSObject(qos:IQoSObject): void
+CheckQoS(): int

Fig. 4. Classes involved in quality of service management used in the default implementation

As the history manager can observe many events, we can define quality of service constraints applying these events. The bounded execution time of video frame processing can be used to demonstrate the implementation of a quality of service object.

```
private string input, output;

private long execTime;

public int CheckQoS() { // bounded execution time

 int delta =
   (int)((TimeSpan)(this.manager
.GetSignalHistoryCurrent(this.output) -
                   this.manager
         .GetSignalHistoryLastOccurrence(this.input)))
                  .TotalMilliseconds;

 // check if delay can be held,
 // if not throw exception

 if (delta > this.execTime) {

  throw new QoSException(
                  "Bounded execution time of "
                  + delta + "ms exceeds limit of "
                  + this.execTime + "ms!");

 }

 return 0; // go on normally

}
```

The quality of service constraints are specified by implementing the interface definition *IQoSObject*. The method *CheckQoS* must implemented which monitors the

QoS constraint. It throws an exception if the constraint cannot be held or slows down execution by returning a numerical value. This values is interpreted as time in milliseconds to wait after the timed assignment. If zero is returned, the code following the timed assignment statement is executed without delay.

The history manager is used to calculate time ranges between events and compare these ranges to specified values. Predefined constraint implementations are provided by a system class library and can be used by the programmer.

```
private string signal;

private int count, delay;

public int CheckQoS() { // throughput

 int delta =
      (int)((TimeSpan)(this.manager
            .GetSignalHistoryCurrent(this.signal)-
                       this.manager
            .GetSignalHistory(this.signal,
                       this.manager
            .GetSignalHistoryCount() - this.count)))
            .TotalMilliseconds;

 // check if delay can be held
 // if not throw exception,
 // otherwise delay execution to

 // reach expected delay value

 if (delta > this.delay) {

  throw new QoSException(
                  "Throughput of " + this.count
                  + " exceeds limit of "
                  + this.execTime + "ms!");

 }

 return delta - this.delay; // slow down execution

}
```

4.4 Results

To justify the concept of quality aware data types it is evaluated against the criteria defined at the beginning of this paper:

1. The expressive power is enhanced because quality of service constraints can be expressed on the data type declaration level. This allows the developer to use embedded QoS without worrying about the implementation. Although the system libraries should include lots of default constraints that can just be used, the developer is enabled to express a new constraint by just implementing a simple interface which emphasizes the extensibility in case of constraint implementation.

Furthermore, one can use polymorphism or other object oriented language concepts to implement, extend, or vary given quality of service constraints. Moreover, we consider the possibility to access a history not only as an aid for the implementation but as something that imposes a basic structure upon QoS monitoring thus improving readability. With the use of a history manager it becomes possible to write programs that are quite close to a specification.

2. The automatic and implicit generation of events triggered at every assignment to the QoS monitored structure increases the safety of the program. Instead of the error-prone task of registering every event manually, this is done by compiler generated code.
3. Optimization of the code can be seen by comparing length and simplicity. Currently no further optimization possibilities can be presented. More investigations are needed to show how the concept of quality aware data types can help to generate more efficient code.

However, our concept is non interruptive in comparison to Esterel. Esterel allows to terminate the execution of a code block prematurely if the result is outdated before its calculation finishes. We claim that this is only a minor restriction. The time gain of immediate cancellation will be small in many cases, whereas subsequently inserting test operations for the case that no sufficient operating system support should be given could slow down the overall performance drastically.

5 Conclusion and Future Work

The aim of this paper is to introduce two new minimal language extensions to improve multimedia application development. New loop statements are presented which can be applied when accessing large multidimensional arrays often used in parts of encoder or decoder software. The extended *foreach* statement is used to allow index access during the loop, and the *forall* loop inherently introduces parallelization to the loop execution. Furthermore the concept of quality aware data types is shown to define QoS monitoring on a data type declaration level which helps us to implement QoS based streaming. These extensions are justified by three basic principles: expressive power, safety and optimization possibilities. This is emphasized by examples of the current implementations in Modula-3 and C#.

Both Modula-3 and C# and their actually used language environments show a lot of pleasant features, and none of the two languages can be declared as a definite winner. Despite of the well-known stylistic differences, the existence of nested procedures in Modula-3 eased the implementation of some features considerably.

In the search towards a multimedia language we want to identify challenges in current programming languages and their embedded concepts. The concepts described in this document still raise a lot questions. Our aim is to define quality of service constraints not directly with code, but with syntax extensions to allow a compact and declarative definition. Furthermore, we want to analyze the possibility of compile time optimizations on code generation based on the compile time knowledge of such constraints.

References

[1] Gordon S. Blair, Jean-Bernard Stefani: Open distributed processing and multimeda. - Addison Wesley Longman Ltd., 1998 ISBN 0-201-17794-3

[2] Svend Erik Knusen: Statement-Sets .- Third International ACPC Conference with Special Emphasis on Parallel Databses and Parallel I/O Klagenfurt, Austria: September 1996

[3] Michael Philippsen, Walter F. Tichy: Modula-2* and its Compilation .- Universität Karlsruhe, 1991 First International appeared in: First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991, Springer Verlag, Lecture Notes In Computer Science 591, 1992

[4] Stefan U. Hänßgen, Ernst A. Heinz, Paul Lukowicz, Michael Philippsen, Walter F. Tichy: The Modula-2* Environment for Parallel Programming, 1993

[5] Michael Philippsen, Markus U. Mock: Data and Process Alignment in Modula-2*, Department of Informatics, University of Karlsruhe, 1993

[6] Laszlo Böszörményi, Carsten Weich: Programming in Modula-3 - An Introduction in Programming with Style. - Springer Verlag, Heidelberg 1996

[7] Mono: Open Source .NET Development Framework - http://www.mono-project.com

[8] Microsoft Developer Network: C# Programmer's Reference, http://msdn.microsoft.com

[9] Jingwen Jin, Klara Nahrstedt: QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy – IEEE Multimedia Magazine, July 2004, pp. 74-87

[10] Ralf Steinmetz, Klara Nahrstedt: Multimedia Systems – Springer Verlag, 2004, ISBN 3-540-40867-3

[11] Standard ECMA-334 - C# Language Specification, 3rd Edition June 2005

[12] Don Box, Chris Sells: Essentials .NET Volume 1 – The Common Language Runtime. - Addison Wsley 2004, ISBN 0-201-73411-7

[13] Greg Nelson: Systems Programming with Modula-3. - Prentice Hall, 1991 ISBN 0-13-590464-1

[14] Zima,H.P., Chapman,B.M.: Supercompilers for Parallel and Vector Computers ACM Press Frontier Series/Addison-Wesley (1990); Japanese Translation, Ohmsha (1995)

[15] J. Z. Li, M. T. Ozsu, and D. Szafron. MOQL: A multimedia object query language. Technical Report TR-97-01, Department of Computing Science, University of Alberta, January 1997

[16] Elmar Stellnberger: Enhancing the Usability of Nested Procedure Values in a Multi Threaded Environment. Manuscript