Towards Adaptive Quality-Aware Programming with Declarative QoS Constraints

MultiMedia C#

Oliver Lampl
olampl@edu.uni-klu.ac.at

Laszlo Böszörmenyi
laszlo@itec.uni-klu.ac.at
Abstract

Quality of service (QoS) has become one of the most important aspects of modern multimedia applications. Nevertheless, programming of QoS-aware, adaptive applications is cumbersome and error-prone. Several QoS-oriented programming methods and tools have been presented, relying on special languages or complex frameworks. Instead, we suggest to slightly extend an existing general-purpose programming language and make thus adaptive programming as easy and as safe as any "normal", exception-based programming.

We extend a general-purpose system programming language (C#) with the following features: (1) A time dimension, which can be added to any existing data type; (2) A new time-constrained assignment (streaming) statement and a time-constrained parameter passing (streaming) mode; (3) Declarative QoS specifications.

We suggest some patterns for adaptive, quality-aware programming and present sample implementations emphasizing the advantages of the new programming model. Performance measurements show negligible overhead when applying the new language features in real-life multimedia applications.

Keywords: Multimedia, Quality of Service, Adaptive Programming
Contents

1 Introduction 1

2 Adaptive Quality-Aware Programming Model 2
  2.1 Time as $n + 1^{st}$ Dimension 2
  2.2 Declarative QoS Constraint 3
  2.3 Compile Time Validation of Constraint Syntax and Semantics 4
  2.4 Streaming statements 5
    2.4.1 Timed (Streaming) Assignment 5
    2.4.2 Timed (Streaming) Method Parameters 6
  2.5 Adaptive Programming 7

3 Adaptive Programming Patterns 7
  3.1 Video Player 7
  3.2 Streaming Server 8
  3.3 WebService Client 9

4 Performance / Optimizations 9

5 Implementation Details 11
  5.1 Quality Aware Variable Declaration 11
  5.2 Declarative Constraint Transformation 11
  5.3 Timed Assignment Transformation 11
  5.4 Timed Parameter Transformation 12

6 Related Work 12

7 Conclusions and Future Work 13

1 Introduction

Multimedia streaming and play-back is the periodical process of dealing with long sequences of data (e.g. video frames) under so-called "soft real-time" constraints. The implementation of this substantial pattern of common multimedia applications is cumbersome and error-prone. The use of additional languages and complex frameworks is needed to achieve quality awareness. Instead of defining QoS and adaptivity as an external aspect of an application we suggest additional features at the programming language level. A set of minimal language extensions allows us to encourage implementation of quality-aware applications.

Our work concentrates on the following feature extensions for a general-purpose system programming language:

- We add an additional dimension to existing data types. This $n + 1^{st}$ dimension represents the time.
- To take use of the time dimension, we introduce a special assignment operation and a new way for passing method parameters.
A new declarative syntax, embedded in a common programming language is used to define QoS specifications.

These features in combination are used to provide common patterns for quality-aware programming. We want to reduce implementation costs and error-proneness by applying these patterns. Our main target is multimedia applications, but the suggested language features can be used in a broader scope. They suit well for all kinds of quality-aware programming where timing constraints need to be held. In particular they can be used to implement Web-Services, Service-Oriented-Architectures [29] or component based systems like J2EE [5].

The rest of the paper is organized as follows. Section 2 presents the adaptive quality-aware programming model describing the new language features in detail. Section 4 demonstrates sample implementations and gives examples of the usage of the presented model. Section 5 discusses performance issues when applying the model in a real-life multimedia application. Section 6 shows some implementation details. Section 7 summarizes related work on this topic. Section 8 closes the paper with contributions and future work.

2 Adaptive Quality-Aware Programming Model

Currently, too many applications avoid being adaptive. Adaptive quality-aware programming is rather hard and complex. In most cases it is handled on the middleware or system level. Our model defines a new structure for programming adaptive quality-aware applications. A part of the control over adaptivity is shifted to the application level. We want to encourage adaptive programming by presenting this new structure. We demerge the code which is in most cases a mixture of time measurements and conditional statements for checking time-based constraints (see Figure 1). The suggested language extensions make adaptive programming safer and easier both on the system and on the application level.

Adaptive quality-aware programming defines a clear structure by separation of concerns. Instead of defining conditional statements we allow declarative definition of QoS constraints. Adaptive code is placed inside an exception handler and is only executed if the application code cannot achieve specified requirements.

2.1 Time as $n + 1^{st}$ Dimension

The concept of quality-aware data types and its underlying framework allow to embed monitoring and control of quality of service definitions directly within a system programming language like C# [2, 27] or
<table>
<thead>
<tr>
<th>QoS Constraint</th>
<th>QL Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>$\forall n,</td>
</tr>
<tr>
<td>Latency</td>
<td>$\forall n,</td>
</tr>
<tr>
<td>Delay Jitter</td>
<td>$\forall n, \delta_1 \leq</td>
</tr>
<tr>
<td>Bounded Execution Time</td>
<td>$\forall n,</td>
</tr>
</tbody>
</table>

Table 1: Common constraints in multimedia systems

Modula-3 [26, 28, 11]. Our implementation is based on Mono [6], an open source implementation of the C# programming language.

A quality-aware data type is defined as an array. Time is the $n + 1^{st}$ dimension, which can be added to any data type. Quality-aware data types are associated with a history which records events. These events are triggered by the timed (streaming) assignment statement and the timed (streaming) parameter passing mode.

// Dynamic QoS variable. Assignment to the
// variable underline a constraint.
int [~] dynamicQoSVariable
  = new int[~(IQoSObject)constraint];

// Static QoS variable. Assignment to the
// variable underline a constraint.
int [~(IQoSObject)constraint] staticQoSVariable;

Quality-aware variables are type-compatible with any variable compatible to their base type. To allow quality checking, we assign constraints to these new data types. Such constraints are either added statically during declaration or assigned dynamically via the new operation. A constraint is represented by the implementation of the IQoSObject interface. This interface defines a method in which the history of quality-aware data types is checked against certain quality conditions. If the condition evaluates to false, an exception is thrown (QoSException). A constraint can be implemented manually or generated by the compiler using a declarative specification syntax.

2.2 Declarative QoS Constraint

The Lancaster University Sumo Project [10] uses QoS specifications defined as annotations on exported interfaces. These annotations specify the QoS offered and the requirements to the environment. These annotations are formally described with the Quality Language (QL). QL represents a formal model for reasoning about time based quality of service (QoS) constraints. It uses events and their histories of occurrences to check whether a given constraint can be met or not.

Table 1 shows the most important constraints used in multimedia streaming applications. The function $\tau$ is called history function and allows access to the history of the events via an index variable. Events are often referenced as $\epsilon$ and an index denotes the type of the event. E.g. $\epsilon_r$ and $\epsilon_s$ symbolize the receiving and the sending event. This allows the representation of lots of different time based constraints e.g. throughput, latency, or delay jitter. Our work relies on QL defined in [10] and is used for specifying timing requirements. These requirements are automatically transformed into compiler-generated code for QoS monitoring.

We define a declarative syntax to specify time-based constraints. This provides a clear and easy-to-use mechanism instead of implementing it manually. The syntax is deduced from the existing mathematical notation of QL by applying the following transformation rules.

Rule 1. Exchange the symbol $\forall$ with the symbol @
Rule 2. Map the history function \( \tau \) to array access of quality-aware variables. Exchange the event name with the name of a quality-aware variable.

Rule 3. Put the expression itself into braces.

Rule 4. Replace variables in the constraint by concrete values or variables defined in your code.

As an example we apply these rules to the definition of throughput. In this example we replace the event \( \epsilon \) with the quality-aware variable named \( \text{var} \). \( \delta \) is set to 40 milliseconds. The utility class \textit{Units} provides absolute time values. It contains static methods to convert expressions into the internally used time representation. The resulting constraint is a definition for throughput of 25 events per second. This is a standard constraint for video streaming where we normally expect 25 frames per second.

```java
// 25 assignments per second
@n {\text{var}[n] - \text{var}[n-25] \leq \text{Units.MSec}(1000)}
```

Note that the expression \([n-25]\) is an abstract specification and not a real array access. The compiler generated code starts checking only at \( n \geq 25 \).

The next example will show how to combine constraints to build more complex QoS specifications. In this example we want to define quality requirements for throughput and jitter defined in one single constraint. This is achieved by combining constraints using logical OR and logical AND combinations. The syntax is similar to defining conditional statements in C#. We combine the above constraint of 25 frames per second with a jitter specification which expects events with a delay between 10 and 50 milliseconds.

```java
// combination of throughput and latency jitter
@n {\text{var}[n] - \text{var}[n-25] \leq \text{Units.MSec}(1000) \land
    \text{Units.MSec}(10) \leq \text{var}[n] - \text{var}[n-1] \leq \text{Units.MSec}(50)}
```

Constraints are not limited to single quality-aware variables. We can define a maximum delay between input and output of streaming data. In this case, the quality language defines events whose histories are compared to each other. The constraint for bounded execution time (see Table 1) uses two different events. \( \epsilon_i \) expresses the incoming event and \( \epsilon_o \), the outgoing.

If the implementation contains two quality-aware variables named \( \text{input} \) and \( \text{output} \), we can limit the bounded execution time of events between input and output to a certain value, e.g. 50 milliseconds.

```java
// bounded execution time
@n {\text{output}[n] - \text{input}[n] \leq \text{Units.MSec}(50)}
```

The presented declarative QoS specifications can be assigned to any declared quality-aware variable. The compiler automatically generates \textit{IQoSObject} classes out of the constraint definition. These classes are instantiated and assigned to the quality-aware variable. The class itself contains an interface method which is called during event triggering to check if the constraint is held or not. The history of the variables is accessed through a QoS management framework from the system class library.

### 2.3 Compile Time Validation of Constraint Syntax and Semantics

The declarative syntax allows automatic validation of syntax and semantic behaviour at compile time. Currently, two approaches exist to verify the semantics of a QoS constraint.

The naive approach relies on calculating an optimal event history at run-time. The optimal event history is calculated using the lower boundaries of constraints. If such boundaries do not exist, almost zero
execution time is used in calculations. The condition is applied to the calculated history. If the condition evaluates to true, the constraint can also be met in a real environment. Otherwise, the constraint is semantically wrong and can never be reached.

The second approach is a formal proof based on static code analysis. Two criteria need to be met.

**Criterion 1.**

\[ x[n-1] \leq x[n] \]

A previous history entry always precedes the current one. We may assume that the total order of events of the same source is guaranteed.

**Criterion 2.**

\[ x[n] \leq y[n] \]

If a timed assignment of a quality-aware variable \( x \) happens before that of \( y \) within the same block, the corresponding history values appear in the same order. This criterion requires sequential execution of the statements of the observed block.

If these two criteria are met, the constraint is semantically not wrong. It cannot be guaranteed that the constraint can be held, but at least it is possible that it evaluates to true.

### 2.4 Streaming statements

#### 2.4.1 Timed (Streaming) Assignment

In multimedia streaming applications two different kinds of value assignments are identified. The first is the usual management operation and does not apply any constraint-checking or quality requirements. This operation is expressed by the "normal" assignment operation (=). In contrast, a streaming operation exists which implements features like video playback. This streaming behaviour is achieved by the timed (streaming) assignment statements.

The timed assignment is responsible for automatic constraint checking and the triggering of events. These events are recorded in the event history of each quality-aware variable. We define the assignment in two versions (~: for write - :~ for read). In the first case, the constraint is defined by the left hand, in the second case by the right hand value. If the constraint cannot be held, an exception is thrown (QoSException). Both directions - read and write - are shown in the following examples.

```csharp
int ~@n {value[n] - value[n-1] <= Units.MSec(10)} value;
// var is assigned to some constraint
int[] array = {1,2,3,4,5,6,7,9};

for (int i = 0; i < array.Length; i++) {
    // Timed writing; Left-hand must be a quality-aware variable.
    value ~: array[i];
    Console.WriteLine("Value: " + value);
}
```

The given examples demonstrates streaming a series of integer values to the screen.

```csharp
int value;
// array is assigned to some constraint
int[] ~@n {value[n] - value[n-1]}
```
<= Units.MSec(10)) array
    = {1,2,3,4,5,6,7,9};

for (int i = 0; i < array.Length; i++) {
    // Timed reading; Right-hand must be a
    // quality-aware variable.
    value := array[i];
    Console.WriteLine("Value: " + value);
}

Both examples finish with the same result. All values of the array were assigned to a single variable
within a loop and displayed on the screen. This was executed under the given constraint specifying a rate
of 100 values per second. In the first example, the quality requirements are specified on the destination
variable and in the second example the definition relies on source variable.

2.4.2 Timed (Streaming) Method Parameters

Another possibility to trigger events are method parameters. This is similar to assigning values to a
variable. We define a new parameter passing mode to distinguish between timed parameters and other
parameter passing modes. In fact, a value can be streamed to a method. This is expressed by the keyword
stream. Stream parameters are passed as values under given QoS constraints. They cause the triggering
of events, which are recorded into the history. The constraint is checked during method invocation. If
the constraint cannot be held, an exception is thrown.

Two possibilities of timed parameter declaration exist. The first is dynamic and uses an object field to
reference the constraint. This value can be changed each time a new instance of the surrounding object
is created.

IQoSObject constraint = ... // some constraint

    // dynamic timed parameter
    void foo(stream object[~"constraint"] input) {
        ... // do something
    }

    // stream data to the method
    foo(stream new object());

The second possibility uses a declarative constraint. To illustrate this we define a method which defines
the maximum delay between subsequent calls not to exceed 100 milliseconds.

    // static timed parameter
    void foo(
        stream object[~@n {input[n] - input[n-1]
            < Units.MSec(100)}] input) {
            ... // do something
    }

    // stream data to method
    foo(stream new object());
2.5 Adaptive Programming

We suggest an adaptive quality-aware programming style, relying on exception handling. In case of QoS violations an exception is thrown. This allows the programmer to explicitly define what should happen when given constraints cannot be held. Either the constraint is changed if it is defined too tight or the programmer redefines the currently executed process to reduce its execution cost.

We use exception handling as a mechanism to indicate that quality requirements cannot be achieved. In C#, checked exceptions do not exist [3]. If the programmer does not catch the exception the application terminates.

Dynamic initialization of quality-aware data types or timed parameters allows adaption of the constraint during runtime. Reinitialization can be used on demand to change QoS parameters. For timed parameters, member variables defining constraints can be accessed within a method declaration. Each time an object of the surrounding class is instantiated a new constraint can be defined. These features allow adaptivity on the constraint level.

Several strategies exist how to implement the adaptational part of an application. In case of video streaming different techniques like rate shaping [13, 23], layered encoding [31], frame dropping [35] or combinations of these [30] can be used. The implementation of the adaptation part is specific to the environment and depending on the used encoders or decoders.

3 Adaptive Programming Patterns

In this section we present sample implementations which emphasize the suggested simple but effective programming model.

3.1 Video Player

Writing a video player is not much effort in a common programming language. Many handy libraries exist which can decode binary input streams frame by frame. A simple way to implement a display component is to define a form, where the display method is overwritten to show a bitmap. Exchanging these bitmaps can be implemented using a small method containing quality-aware method parameters.

```csharp
private Bitmap frame;

// set new frame for display
void SetFrame(
  < Units.MSec(1000)) frame) {
  this.frame = frame;
  this.Refresh(); // causing display of frame
}
```

Every time this method is called it causes the current display to be refreshed using a new bitmap passed as parameter. As long as the defined constraint can be held the video playback runs normally. In case of a QoS violation, the method throws an exception and allows the programmer to react within the decoding process to speed up the performance. Here the adaptational behaviour can be implemented by throwing away frames or lowering the resolution of the decoded frames. The adaptivity itself relies on the possibilities of the used encoding/decoding framework. The quality-aware method itself does not provide adaptivity, rather it enables to embed it easily.

```csharp
// decode a frame
```
void DecodeFrame() {
    // decoding process
    Bitmap frame = GetDecodedFrame();

    try {
        form.SetFrame(stream frame);
    } catch (QoSException e) {
        // lower resolution
        // speed up decoding process
        ... // apply code for adaptation
    }
}

Depending on the used decoder, different strategies can be used to achieve an optimal display process. The compiler-generated code ensures that QoS violations are detected automatically.

3.2 Streaming Server

The implementation of a simple streaming server can be extracted out of most network communication examples. We do not concentrate on file organization, packetization, or multi-threading issues. In general quality-aware programming can be applied everywhere in the source code. In case of streaming servers network throughput is very important. The timed assignment supports to ensure that a given throughput is held over an existing network connection.

private Socket client; // client reference
private bool inTransfer = true;
// timed variable
private Byte[][] byteData
    = new Byte[][] {
        {byteData[n - 1] - byteData[n] < Units.MSec(50)}
    } byteData;

// prepare single block
private Byte[] PrepareData() {
    // construct binary data for transfer
}

// transfer binary data over the network
private void Transfer() {
    // as long as data exists
    while (inTransfer) {
        try {
            // stream prepared data under constraint
            // over the network
            byteData ^= PrepareData();
            client.Send( byteData,
                         byteData.Length,
                         0 );
        } catch (QoSException e) {
            // current constraint was not held
            AdaptData();
        }
    }
}
A method call is responsible for preparing data blocks of fixed length. These data blocks are transferred over a socket. A quality aware variable is used to ensure that a specified bit-rate is held. If the time needed for transfer exceeds the defined boundaries an exception is thrown. The exception handler calls a method to adapt the data to be transferred or changes the constraint to a value which is achievable.

3.3 WebService Client

To show a different example, let us access a webservice using timed parameters. The following example [4] shows a simple WebService implementation, which converts temperature values from Celsius to Fahrenheit.

```java
public class TemperatureService

    public double ConvertTemperature(
        double dFahrenheit) {
        return ((dFahrenheit - 32) * 5) / 9;
    }
}
```

The client code accesses the WebService through a generated proxy class (MyTemperatureService). The stub is created out of the WSDL [8] specification. In our example, we want to call the temperature service every second.

```java
// constraint used for convert method
private IQoSObject wsConstraint =
    @n {dFahrenheit[n] - dFahrenheit[n-1]
     <= Units.MSec(1000)};

// does webservice call
public double convert(
    /* dynamic stream parameter */
    stream double[~wsConstraint] dFahrenheit) {
    // initialize proxy object
    MyTemperatureService ws
        = new MyTemperatureService();
    // call the webservice
    return ws.ConvertTemperature(dFahrenheit);
}
```

The client, e.g. a sensor application, calls the method to convert the temperature values. QoS is achieved by using timed parameters. Each successive call has to be completed within the specified range of time (1000 milliseconds). If this timing behaviour cannot be held an exception is thrown.

4 Performance / Optimizations

It is obvious that an enriched assignment operation cannot compete with the standard assignment, which in most cases can be mapped to a single CPU instruction. Instead of execution a single statement additional functionality is executed by a framework in the runtime system. The overhead produced is approximately one order of magnitude compared to the original operation. Thus, the question arises: Is it feasible to use the presented features at all?
In a typical application we have to replace only a tiny portion of *normal* assignments by streaming. Thus, the effective overhead caused by adaptive programming is much less than might be expected at first. The performance overhead is investigated by splitting up the execution costs of a standard video decoding process. In [32] we showed the amount of time used for the decoding process of a H.264/AVC [1] implementation. One of the most expensive parts of the overall process is the inverse transformation which needs more than 40% of the time.

When embedding our new language features for quality aware programming it produces only an overhead less than 1% of the whole process. We added the timed assignment statement to assign fully reconstructed frames to a display variable. The time needed for one single streaming statement was approximately 750 ticks. Executing the application with a required frame rate of 25 frames per second causes an overhead of approximately 2 milliseconds.

Several measurements have been done to show the performance impact of the new language features. *Figure 2* displays the amount of time needed for executing the timed assignment with different numbers of constraints. All measurements were made on a SMP multiprocessor machine with 2 CPUs and 2GB RAM, running *Linux*. The time needed for constraint checking increases linear with the number of constraints assigned to a variable. The absolute time is measured in ticks. A millisecond is equal to 10000 ticks.

The amount of constraints only produces minimal overhead. Does the size of the event history has an impact on the performance? *Figure 3* displays the results of varying the number of elements stored in the history. Again, the execution time increases linear with the number of elements stored in the history. The initial filling of the history causes a lot more time than the later processing. This is caused by memory allocation at the beginning. After the number of elements in the history has reached the defined limit a peak is identified. This peak is caused by the first element removal out of the history.

Although the absolute values presented seem to be quite low, quality checks on the millisecond level may not work as the order of magnitude for the execution of the check reaches the constraint value itself. The first problem is the initial filling of the history and memory allocation. Special handling of this initial phase needs to be implemented. Another possibility to speed up performance is to reduce the number of events put into the history. Of course this could harm the level of adaptivity and reactivity of the code, but it can reduce the execution costs. More research needs to be done in this area to provide a concrete concept on how an optimal event history can be built based on the compile time knowledge of constraints.
5 Implementation Details

In this section some details about the implementation of the presented programming model are discussed. The implementation was done using the Mono [6] framework, which is an open-source implementation of the .NET framework [7]. The historization of events is handled via a QoS framework which is part of our runtime environment.

The QoSHistoryManager is the main component of this framework. This component is responsible for registration of quality-aware variables and their assigned constraints. During the timed assignments this component builds event histories and provides an interface to access these entries for a specific point in time. The entries are retrieved for constraint validation. The compiler generates code for automatically accessing the framework in the runtime by taking advantage of the new special statements.

5.1 Quality Aware Variable Declaration

The declaration of a quality aware variable is similar to that of a standard variable. Additionally, the variable is registered in the QoS framework and a constraint is assigned. This is handled by compiler generated code.

5.2 Declarative Constraint Transformation

Constraint declaration allows an easy transformation to conditional statements. These statements are embedded into a class implementing the IQoSObject interface. The array access to the history of events is mapped to calls to the QoS framework. The time is measured using DateTime objects. The calculation itself is done in milliseconds.

5.3 Timed Assignment Transformation

The timed assignment executes three functional blocks. The first block is responsible for creating the event history. This is implemented as a simple method call to the framework behind, which captures all calls up to predefined limit. The limit may vary on the used constraints and their requirements to the history.

The second block is the execution of the constraint check. This is done by retrieving the currently assigned constraint object and calling a predefined interface method. This method can then return
normally or cause an exception in case of QoS violations.

The last block is responsible for the assignment itself and delegates the assignment to the original implementation.

5.4 Timed Parameter Transformation

The timed parameter mode is implemented via the same functional blocks as the timed assignment. The generated code for this functionality is inserted at the beginning of the method containing timed parameters. Additionally an initialization block is created. It ensures that the defined constraints are assigned to the timed parameters within the QoS framework.

6 Related Work

A lot of research has been done in the field of quality of service (QoS). Many research papers can be found focusing on specifying and negotiating QoS requirements in all fields of computer science. A variety of different notations exist to express QoS-related mechanisms like resource reservation, admission control, and adaptation. Jin and Nahrsted [24] present a classification of existing QoS specification languages. These languages are defined on application-, user- or resource-level and try to cover most QoS aspects within a user friendly notation. QoS modeling languages like QML [17] or CQML [9] focus on specifying QoS contracts between different components.

Many research was put into investigating middleware systems to enable the implementation of distributed multimedia [34, 19, 15, 25, 10, 33]. Such middleware is able to take over QoS concerns from the application level. Dynamic configuration of the QoS management is required. Resource allocation and correct distribution of requests are hidden by the system. In contrast, we shift the control over adaptivity to the application level.

QoS is required for different services. Multimedia and real-time environments are not the only target of QoS. Distributed service oriented architectures show increasing demand for QoS e.g. in the field of Web-Services [18] as well as in mobile communications. Streaming multimedia content over wireless networks claims for adaptivity. [16] studies the issues of QoS provisioning in such systems. In contrast to QoS in network communication, [14] recommends splitting application level QoS to each individual process of the application. They recommend to distribute the overall QoS specification over the individual components and demonstrate their approach with the help of a MPEG2 video decoder application.

In [12], an architecture is presented which allows streaming of multimedia content with adaptive QoS characteristics. The implementation of adaptive streaming applications requires mechanisms to monitor network conditions and adapt the transmission of data. The presented work focuses on monitoring network conditions and estimating the appropriate rate for transmission of data. We embed the monitoring aspect on the application level.

[22] present the design of a QoS aware email application. It demonstrates a flexible model for using simple declarative requirements statements. In [21], a declarative characterisation to define profile models for expressing QoS requirements is used. These requirements are formulated as formal expressions defined in [20]. Expressions are constructed through combinations of atomic expressions. The declarative syntax allows to check the correctness and consistency of defined rules. In contrast to our work, it focuses mainly on the definition and negotiation of all kinds of quality requirements. We provide a programming model to implement QoS monitoring based on specifying time based constraints.
7 Conclusions and Future Work

We have presented a new programming model for developing adaptive quality-aware applications. The programming model is based on the concept of quality-aware data types. We demonstrated how to construct declarative constraints which are applied in combination with a new timed (streaming) assignment statement and timed (streaming) parameter passing mode. Patterns show how the new features can be used to implement exception based adaptive programs. The execution costs for the new operations are analyzed based on detailed measurements.

We need further investigation how to build an optimal event history. The time for initialization has to be reduced. The compiler based semantic analysis of constraints needs to be implemented as well. We want to investigate in detail the issue of correctness proof of constraints. Furthermore, we want to analyse how the compile time knowledge of QoS constraints can improve the efficiency of the code. One possibility is to give language support for enabling the compiler to emit code for special multimedia instructions, as provided by some modern CPUs.

One of the next milestones is the implementation of a QoS benchmarking tool. This tool will do analysis of existing systems. It will provide a set of QoS constraints which are achievable on a given platform. This information can be used to indicate if required performance and reliability for e.g. video decoding, etc. can be reached or not.

References


Institute of Information Technology
University Klagenfurt
Universitätsstr. 65-67
A-9020 Klagenfurt
Austria
http://www-itec.uni-klu.ac.at