MultiMediaC# - QoS-Aware Programming with C#

Oliver Lampl, Laszlo Böszörmenyi

Institute of Information Technology, University of Klagenfurt, Austria
olampl@edu.uni-klu.ac.at, laszlo@itec.uni-klu.ac.at

Abstract. Providing QoS-awareness and adaptive behavior in multimedia applications is cumbersome and errorprone. Many different frameworks and even additional languages exist which try to support QoS-aware application development. In many cases this is done by shifting away QoS processing facilities from the prgrammer and hiding them inside resource layers or other middle-ware. Instead, we integrate QoS directly into the common programming language C#.
In this paper we introduce the specification of MMC# as an extension to the C# programming lanugage focusing on adaptive, QoS-aware programming. The new language features provide a solution to the three common problems in the field of QoS processing: (1) constraint declaration, (2) constraint monitoring and (3) providing adaptive behavior in multimedia applications. Furthermore, declarative definition of QoS requirements directly within the programming language allows to apply semantic constraint analysis - partly at compile-time, partly run-time - to check the correctness of specified requirements and further provide optimizations by automatically removing not required constraints. Last but not least, this paper describes the structure of the Mono compiler which has been the basis for development. We illustrate how the compiler has been modified to support these novel language extensions and the semantic analysis of QoS constraints.

1 Introduction

Up to the present, the software engineering community has largely neglected the problems surrounding the development of multimedia applications that dynamically adapt their behavior to both the required Quality of Service (QoS) and to available resources. QoS and adaptation of course have been recognized as important issues and have been subsequently discussed in the relevant literature, but these discussions have usually restricted themselves to a limited context and rarely concentrated on the design issues of QoS aware, adaptive applications.

Many different frameworks exist which promise support for QoS handling and adaptivity which is achieved by defining QoS contracts within separate specification languages. Furthermore, a lot of multimedia frameworks and middleware systems have been implemented which claim to provide QoS handling. However, their QoS capabilities are often limited to a small context or even completely hidden to the programmer.

Currently, there is no support for QoS-aware programming within common general purpose programming languages like C# or Java. We introduce an extension to the programming language C#, called MMC#, which focuses on adaptive QoS-aware programming. MMC# extends current C# programming capabilities by adding the following features:

- Constraint declaration - A declarative syntax is used to provide QoS constraint specification.
- Constraint monitoring - Timed data types utilize the $n + 1^{st}$ dimension as representation of the time and provide historization with the help of special assignment operations. These timed assignments apply automatic time recording and QoS constraint evaluation.
- Adaptivity - In case of QoS violations exception are thrown, which enable adaptive programming to be as easy as any other kind of exception handling.

Furthermore, we apply semantic analysis on declarative constraint specification to provide their correctness and even optimize their representation. This analysis is splitted into two parts:

- Compile-time Analysis - The compiler analyzes the constraints and provides correctness checks and optimizations for constant declarations. In case of dynamic values it generates code which is executed during constraint initialization.
- Run-time Analysis - Constraint declaration can use dynamic elements like parameters or method calls which are not available during compile-time. These values are analyzed during run-time and in case they lead to an incorrect constraint definition, an exception is thrown.

MMC# and the semantic analysis have been implemented within the Mono C# compiler. The Mono framework is an open-source implementation of the .NET framework and provides portability of .NET applications to other platforms than Windows. We briefly describe the structure of the compiler and show how the presented extensions have been integrated.

2 MMC# Specification

Keywords

```
constraint
stream
```

Types

```
type:
        timed-type
        value-type
        reference-type
        type-parameter

timed-type:
        non-timed-type [ ~ constraint-reference_opt ]

non-timed-type:
        value-type
        reference-type
```

Assignments

```
assignment:
        unary-expression assignment-operator expression
        unary-expression ~: expression
        unary-expression :~ unary-expression
```

Methods

```
parameter-modifier:
        ref
        out
        stream
```

```
argument:
          expression
          ref variable-reference
          out variable-reference
          stream expression
```

Classes

```
class-member-declaration:
          constant-declaration
          parameterizable-constraint-declaration
          field-declaration
          method-declaration
          property-declaration
          event-declaration
          indexer-declaration
          operator-declaration
          constructor-declaration
          finalizer-declaration
          static-constructor-declaration
          type-declaration


parameterizable-constraint-declaration:
          constraint-modifiers_opt constraint identifier
(formal-parameter-list_opt) => constraint-declarations ;


constraint-modifiers:
          constraint-modifier
          constraint-modifiers constraint-modifier


constraint-modifier:
          private
          protected
          public
          static


constraint-declarations:
          or-constraint-declaration


or-constraint-declaration:
          and-constraint-declaration
          or-constraint-declaration ||
and-constraint-declaration


and-constraint-declaration:
          constraint-declaration
          and-constraint-declaration &&
or-constraint-declaration


constraint-declaration:
          constraint-declaration
          and-constraint-declaration &&
or-constraint-declaration
```

Constraint declaration

```
constraint-declaration:
          @ identifier { constraint-condition-with-exception
}


constraint-condition-with-exception:
          or-constraint-condition
          or-constraint-condition : throw-statement


or-constraint-condition:
          and-constraint-condition
          or-constraint-condition || and-constraint-condition


and-constraint-condition:
          constraint-condition
          and-constraint-condition && or-constraint-condition


constraint-condition:
          qos-expression
          qos-expression qos-conditional-operator
qos-expression


qos-conditional-operator:
          <
          <=
          ==
          >
          >=


qos-expression:
          qos-primary-expression
          qos-expression qos-math-operator qos-expression


qos-math-operator:
          +
          -
          *
          /
          %


qos-primary-expression:
          primary-expression
```

2.1 Types

The type definition[1] is extended to support timed-types. Timed-types are value-types and similar to array-types.

A timed datatype in this context also called quality-aware data type is defined similarly to an array type. 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 (see section 2.2) and the timed (streaming) parameter passing mode (see section 2.3).

Quality-aware variables are type-compatible with any variable compatible to their base type. During type-checking the additional dimension is omitted. To allow QoS checking, constraints are associated to timed data types. Such constraints are of type `System.IQoSObject` and are either added statically during declaration or assigned dynamically via the `new` statement.

[1] This block refers to section 11 page 107 and following of the C# Standard [1].

2.2 Assignment operators

Two new assignment operators[2] are added. The so called timed-assignment or streaming-assignment assigns a value from or to a timed-variable. This operation is done under QoS control. It advises the compiler to add QoS monitoring code.

Each time the assignment is executed the timed-variable records an event and the assigned constraint is evaluated. If the evaluation fails, an exception is thrown. This behavior is explained in section 2.5 in detail.

A write-timed-assignment requires the left part of the assignment to be a timed-type. Accordingly, a read-timed-assignment requires a timed type on the right side.

Statically assigned constraints are initalized when declaring a variable. Such constraints cannot be changed over the life-time of the variables they are associated. Instead, dynamic constraint assignments are applied using the `new` operator.

In multimedia streaming applications two different kinds of value assignments exist. The first is the usual management operation and does not need any constraint-checking or quality requirements. This operation is expressed by the "normal" assignment operation (`=`) which is executed best effort. In general, no specific timing requirements exist for management operations. In constrast, a streaming operation exists which implements features like video playback. This streaming behavior is achieved by the timed (streaming) assignment statements and requires QoS-aware behavior.

2.3 Methods

A further parameter passing mode[3] is added. `Stream` parameters are passed by value, but with constraint evaluation. Accordingly, a streamed argument is denoted with the keyword `stream`. Timed parameters require a timed data type within the declaration of the parameter list. The QoS constraint is attached when the method is called for the first time. In case of static methods only statically assigned constraints can be used.

2.4 Classes

The class member declaration[4] is extended to provide parameterizable constraints. Parameterizable constraints allow the definition of reusable constraints by specifying constraint templates as class members via the keyword `constraint`.

Constraints can be combined by logical operators `&&` and `||`. This allows building complex constraints out of simple ones, e.g combining throughput and jitter constraints. In case of a QoS constraint violation, an exception is thrown. The semantic of logical concatenation of using the operator `&&` is that every

[2] This block refers to Section 14.14 page 218 and following of the C# Standard [1].
[3] This block refers to Section 17.5.1 page 287 and following of the C# Standard [1].
[4] This block refers to Section 17.2 page 269 and following of the C# Standard [1].

constraint needs to be met. In contrast, when using the operator `||`, one single constraint satisfaction is enough to fulfill the whole constraint.

Parameterizable constraints allow QoS constraints to be reused. Such constraints are declared in the same way as other class members and make use of the same accessibility features. Furthermore, they allow the definition of parameters which can then be used inside the constraint. During constraint instantiation actual parameters are mapped to the formal parameters in the constraint definition.

2.5 Constraint declaration

Constraints are declared similarly to the mathematical forall expression ($\forall$). The identifier symbolizes a control variable which is incremented each time the constraint is evaluated. The initial value is 0, which means that no timed assignment was executed. The current value of the control variable represents the current event. This identifier can be used to access the timing history of a timed-type. Only past events can be accessed.

The constraint condition evaluates to a boolean value. If the resulting value is `false`, the constraint is violated and an exception is thrown.

A throw statement[5] can be optionally used to throw user defined exceptions in case of QoS violations. If no exception is specified, a `System.QoSException` is thrown.

Constraint conditions can be combined by logical operators `&&` and `||` which allows building complex logical expressions. Such complex conditions are evaluated as one single block.

A constraint condition is required to be of a type that can be implicitly converted to a boolean value. It allows a set of comparison operators to be used.

A `qos-expression` is used to apply mathematical operations within parts of constraints. It is clearly defined which operations are allowed to reduce complexity of constraint declaration.

Primary expressions[6] are the simplest forms of expressions. They allow e.g. variable access, paranthesis and array element access. This array element access is applied to access the timing history of a timed variable within a constraint declaration.

A constraint is represented by the implementation of the *System.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 (`System.QoSException`). A constraint class can be implemented manually or generated by the compiler using the explained declarative specification syntax.

Furthermore, a constraint is only valid if there is enough information in the history. If the history does not contain enough events, the declared constraint cannot be checked and is always evaluated to true. The event counter which is declared using the symbol @ is used to access the current history entry.

[5] This block refers to Section 15.9.5 page 243 of the C# Standard.
[6] This block refers to Section 14.5 page 165 of the C# Standard.

3 Semantic Constraint Analysis

In some cases, complex constraint definitions may be hard to understand and mistakes can occur. Correctness checks executed by the compiler and by compiler generated code at run-time help the application developer to define correct constraints by applying semantic analysis and also provides optimizations by removing unnecessary constraints.

The declarative syntax allows automatic validation of syntax and semantic behavior at compile time. Syntactic correctness is achieved by correct grammar definition. The semantic analysis is divided into two parts:

1. Compile-time analysis
2. Run-time analysis

The first part is executed during compile-time and evaluates constant expressions which are subsequently compared to each other. Depending on the result of the comparison, the compiler optimizes the constraint or generates a compile-time error message or warning. If the constraint contains dynamic elements like parameters or method calls, an additional code is generated to carry out the check during run-time.

The second part of the compiler generated semantic analysis is performed at run-time, during constraint instantiation. All parameter values are evaluated against the same criteria as constant expressions during compile-time. In case of inconsistent constraint definitions, an exception is thrown. This can be compared to array bound checking [3] which is used in mainstream programming languages like C# or Java [4].

Although semantic analysis examines constraints at compile-time and during instantiation, some uncertainty of correctness remains when using user defined methods. Adding method calls allows constraints to be extensible for any required behavior, but they can also harm a constraint definition. A method call may last too long and the constraint check itself may take longer than the action being monitored. Furthermore, such method calls may lead to erroneous behavior which is eventually not captured by exception handling code. The current implementation of semantic analysis allows user defined methods, but does not analyze them in detail, except methods defined in the system class `QoS.Units`.

After the constraint analysis has been completed, a matrix with relations between constraints and parts of constraints is available. These relations help the compiler making decisions about possible optimizations. A relation between two constraints in the matrix can be expressed by one of the following types:

- **Weaker** (W) - A weaker constraint is logically implied by a stronger one.
- **Equality** (E) - Two constraints express an equivalent behavior.
- **Stronger** (S) - It is the inverse relation to the *Weaker* relation.
- **Inconsistent** (I) - Inconsistent constraints represent contradictions. Such constraints will never evaluate to true.
- **Different** ($-$) - Two constraints are different and thus both are required to express the specified requirement.

– **Dynamic Constraint** (D) - The constraint uses dynamic elements and cannot be evaluated at compile-time. Dynamic elements are method calls or class member accessses which result in varying values during run-time.

The constraint analysis process consists of the following steps:

1. Split constraint definition into conditional blocks.
2. Convert condition blocks into normalized form.
3. Compare each normalized constraint condition to each other and fill evaluation matrix.
4. Decide constraint reconstruction based on evaluation matrix to provide optimized code or warning and error messages.

3.1 Constraint Splitting

Concatenated constraints are too complex to be analyzed. Therefore, the whole constraint is split into its conditional parts. Time independent constraint parts are omitted. These parts are not covered by the constraint analysis.

An example is used to illustrate the constraint analysis. It analyzes the following QoS requirements which are expressed in Listing 1.1:

– A frame rate of 25 frames per second should be used for playback.
– The delay between two consecutive frames should not exceed 100 milliseconds.
– The deviation of delay between two consecutive frames should not exceed 20 milliseconds with respect to the frame rate of 25 frames per second.
– A group of 5 frames needs to be processed within 500 milliseconds.

```
1 @n{frame[n] - frame[n-25] < QoS.Units.Sec(1)} &&
2 @n{frame[n] - frame[n-1] < QoS.Units.MSec(100)} &&
3 @n{   frame[n] - frame[n-1] > QoS.Units.MSec(20)
4    && frame[n] < frame[n-1] + QoS.Units.MSec(60)} &&
5 @n{frame[n] - frame[n-5] < QoS.Units.MSec(500)}
```

Listing 1.1. Declarative QoS Constraint Specifying QoS Requirements.

After constraint splitting is finished, the following conditional blocks have been identified (Table 3.1).

Part	Condition
C_1	`frame[n] - frame[n-25] < QoS.Units.Sec(1)`
C_2	`frame[n] - frame[n-1] < QoS.Units.MSec(100)`
C_{3a}	`frame[n] - frame[n-1] > QoS.Units.MSec(20)`
C_{3b}	`frame[n] < frame[n-1] + QoS.Units.MSec(60)`
C_4	`frame[n] - frame[n-5] < QoS.Units.MSec(500)`

Table 1. List of Extracted Conditions.

3.2 Constraint Condition Normalization

To make constraints comparable they need to be transformed into a normalized form (Table 3.2). All constraints are converted into one of these forms (1). If a constraint condition cannot be converted into one of these forms, it is considered to be different.

$$\begin{aligned} &\forall n, \tau(\epsilon, n) - \tau(\epsilon, n-k) < \delta \\ &\forall n, \tau(\epsilon, n) - \tau(\epsilon, n-k) = \delta \\ &\forall n, \tau(\epsilon, n) - \tau(\epsilon, n-k) > \delta \end{aligned} \tag{1}$$

Part	Condition
C_1	`frame[n] - frame[n-25] < QoS.Units.Sec(1)`
C_2	`frame[n] - frame[n-1] < QoS.Units.MSec(100)`
C_{3a}	`frame[n] - frame[n-1] > QoS.Units.MSec(20)`
C_{3b}	`frame[n] - frame[n-1] < QoS.Units.MSec(60)`
C_4	`frame[n] - frame[n-5] < QoS.Units.MSec(500)`

Table 2. List of Normalized Conditions.

3.3 Constraint Condition Comparison

Constraint conditions are compared based on the defined normalized form. The parameters k and δ are the condition parameters. They can be a constant value or defined as a parameter in case of parameterizable constraints. If they are constant, they can be used for further investigations. In case they are passed as parameters they are marked as a *dynamic constraint* (D).

If constraint parts use the same timed variables, they can be compared. If they use different timed variables, it is not possible to compare them.

$$R_{A\to B} = \begin{cases} E, & \text{if } \delta_A = \delta_B \wedge k_A = k_B \\ W, & \text{if } op_A = op_B = \begin{cases} <, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} \geqslant 0 \wedge k_A \geqslant k_B \\ =, & \text{if } \frac{\delta_A}{k_A} > \frac{\delta_B}{k_B} \\ >, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} < 0 \wedge k_A < k_B \end{cases} \\ S, & \text{if } op_A = op_B = \begin{cases} <, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} \leqslant 0 \wedge k_A \leqslant k_B \\ =, & \text{if } \frac{\delta_A}{k_A} < \frac{\delta_B}{k_B} \\ >, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} > 0 \wedge k_A > k_B \end{cases} \\ D, & \text{if } \delta_A \vee \delta_B \vee k_A \vee k_B \text{ are dynamic} \\ I, & \text{if } op_A \neq op_B \wedge \begin{cases} <>, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} < 0 \\ ><, & \text{if } \frac{\delta_A}{k_A} - \frac{\delta_B}{k_B} > 0 \end{cases} \\ -, & \text{else} \end{cases} \tag{2}$$

Condition relations are evaluated by comparing k, δ and the comparison operator (2). The method call to the class `QoS.Units` is not important in this case, due to the fact that the method body of the static methods only contain formulas with constant values. The relation matrix is the result of the comparison of these conditions. Table 3.3 illustrates the complex relations between the single conditions.

	C_1	C_2	C_{3a}	C_{3b}	C_4
C_1	E	-	-	-	-
C_2	-	E	-	W	S
C_{3a}	-	-	E	-	-
C_{3b}	-	S	-	E	S
C_4	-	W	-	W	E

Table 3. Relation Matrix.

3.4 Constraint Reconstruction

```
1 @n{frame[n] - frame[n-25] < QoS.Units.Sec(1)}
2   &&
3 @n{frame[n] - frame[n-1] > QoS.Units.MSec(20)
4       && frame[n] < frame[n-1] + QoS.Units.MSec(60)}
```

Listing 1.2. Reconstructed QoS Constraint.

The relation matrix allows reasoning about the importance of specific conditions (Table 3.4). If the matrix (Table 3.3) contains equal conditions other than self equality (main diagonal), these conditions can be reduced to one single representative. Weaker conditions can be omitted and only stronger ones are taken into account. Different conditions are important to preserve the behavior of the whole constraint. Inconsistent constraint declarations cause the compiler to produce error messages and the compilations fails. If the constraint is modified by the compiler, warnings are produced which inform the developer about the inefficient definition of the constraints.

```
1   @n{   // delay of 30 milliseconds
2   input[n] - input[n-1] < QoS.Units.MSec(30)
3      && // frame rate of 25 frames per sec
4   input[n] - input[n-25] < QoS.Units.MSec(1000)}
```

Listing 1.3. Weaker Constraint Example.

If dynamic conditions are encountered, the compiler generates check code which is executed at run-time during constraint initialization. This check code compares passed parameter values with other parameters or constant values defined within the constraint. If the check fails an exception is thrown (`System.QoSInitializationException`).

Based on the information within the relation matrix a new constraint can be built.

Relation	Description
W	If one constraint is semantically weaker than another, the weaker one can automatically be removed. A weaker constraint is logically implied by a stronger one. Listing 1.3 presents an example of this. If the delay between two consecutive frames never exceeds 30 msec then this implies that the frame rate is below 25 fps (it is actually 33,33 fps).
E	If two constraints are equivalent, one can be removed.
S	If one constraint is semantically stronger than another, the weaker one can be removed automatically.
I	If two constraints contradict each other, a compile-time error is generated. Listing 1.4 illustrates that either the first evaluates to true and the second fails or the second evaluates to true and the first fails. Such a constraint is obviously erroneous and should be removed.
–	Two constraints are different and thus both are required to express the specified requirement. No action is required.
D	The constraint uses dynamic elements and cannot be evaluated at compile-time. Dynamic elements are method calls or class member accesses which result in varying values during run-time. Check code is generated which evaluates constraint correctness during initialization.

Table 4. Relation-based Actions.

```
1  @n{ // maximum delay of 30 milliseconds
2  input[n] - input[n-1] < QoS.Units.MSec(30)
3     && // minimum delay of 50 milliseconds
4  input[n] - input[n-1] > QoS.Units.MSec(50)}
```

Listing 1.4. Inconsistent Constraint Example.

4 Extending the Mono C# Compiler

This chapter describes briefly the structure of the monon C# compiler and how modifications are applied.

5 Mono Project

The Mono project [5, 6] is an open source implementation of the .NET platform mainly developed for UNIX, but also supports Windows and Mac OS X. Its aim is to provide a cross-platform for .NET applications to enable windows applications to run on UNIX based systems and Mono developed applications to run on Windows based systems. Mono includes the following components:

- **Common Language Infrastructure** - The CLI is a virtual machine which contains a class-loader, JIT (just-in-time) compiler and GC (garbage collector) infrastructure.
- **Class Library** - The provided class library is .NET compatible and also includes Mono-provided classes.

– **C# Compiler** - The C# compiler is the main compiler of the Mono framework. There is already support for other languages like Visual Basic, Oberon or Object Pascal.

The .NET framework uses a virtual machine like environment called the Common Language Runtime (CLR) and is intended for use with several different programming languages. The .NET platform has compilers [9] that target the virtual machine from a number of languages: Managed C++, Java Script, Eiffel, Component Pascal, just to name a few. The CLR and the Common Type System (CTS) provide a common run-time to all of these languages. Independent of their syntax and language structures the target output is the Common Intermediate Language (CIL) which allows interoperability between all CLI compatible languages. When a CIL application is executed, the code is compiled to native machine code for the appropriate architecture.

6 Structure of the Mono C# Compiler

The Mono C# compiler is one central aspect of the Mono framework. It is an ECMA-Standard [1] compliant implementation of a compiler for the C# programming language targeting the standard compliant CLI [2]. Each new version of the programming language is provided via a separate compiler executable.

– **mcs** - A C# compiler targeting .NET 1.x framework.
– **gmcs** - A C# compiler targeting .NET 2.0 and 3.5 framework.
– **smcs** - A C# compiler targeting .NET 2.1 framework including Silverlight APIs.
– **dmcs** - A C# 4.0 compatible compiler (currently under development).

The implementations of MMC# started with implementing the language extensions with mcs. Later on the development moved to gmcs.

Mono C# compiler [7] is completely written in the C# programming languages and makes heavy use of .NET API calls inlcuding `System.Reflection` and `System.Reflection.Emit`.

The compilation process is divided into the following parts:

1. **Parsing** - The compiler parses a set of source files. This is done with the JAY parser which is a port of Berkeley Yacc to Java that was later ported to C# by Miguel de Icaza. Additionally used assemblies are loaded after parsing has been completed.
2. **Type Hierarchy Processing** - After parsing has finished the type hierarchy is resolved and populated to the type system. After this step the program skeleton is complete.
3. **Code Generation** - This phase is divided into two parts. The first part is responsible to provide semantic analysis and check if the code is correct. The second one really emitts the code. After executing code generation the output is saved to the disk.

[7] provides a detailed overview about the file organization of the compiler and explains the responsibilities of each code block.

6.1 Lexical Analysis

During the first phase of the compiler execution all input is processed by the tokenizer. The tokenizer splits the input text into small parts which then are used by the JAY parser for further processing.

If the tokenizer returns a token, its location is recorded within a property which is accessible by the parser. This information is used to provide correct error messages pointing to the location of the problem including line number and column. If the location cannot be determined by the compiler, error messages pointing to line 0 are produced which represent anonymous error messages.

C# has only limited pre-processing capabilities compared to the programming language C [10]. A separate method is invoked when detecting pre-processing directives which start with the symbol #.

6.2 Syntax Analysis

The JAY parser takes the previously generated tokens and evaluates the syntax of the source program. The grammer of the parser does not exactly match the grammar provided within the standard, because the standard has been written for human beings and not for machines.

Each statement or expression identified by the parser is represented by a class. If the parser finds an appropriate token sequence, it instantiates an object of the corresponding class representing the analyzed language construct.

The output of the syntax analysis is an abstract syntax tree (AST) which contains expression trees and all statements required for code generation. It further allows semantic analysis to check the correctness of the code and evaluation of all used types.

6.3 Semantic Analysis

The abstract syntax tree is the internal representation of the provided program. It is the input for the semantic analysis. This analysis starts by resolving the interface hierarchy. Next, classes, structures, constants and enumerations follow. After processing the main building blocks of applications methods, indexers, properties, delegates and events are resolved. At the end of this process elements containing code are checked to be semantically correct.

6.4 Code Generation

The code generation is done with the help of context objects (`EmitContext`) which keep track of current namespaces, type containers and the state of code generation. The resolved abstract syntax tree is the basis for code generation. With the help of the `ILGenerator`, the program is translated into the Common Intermediate Language (CIL).

7 Embedding Language Extensions

Depending on the required enhancements, several different possibilities exist to modify the existing Mono C# compiler. The implementation of the language features presented within this paper have been applied to gmcs which is the compiler version targeting the .NET framework 2.0.

The tokenizer is based on a manual implementation and needs to be modified accordingly in case new tokens are required. For the current extensions this was made for the new assignment statements and keywords introduced within MMC#.

The parser is implemented as a JAY-parser specification which is a mixture of grammar definitions and related C# code which is executed when a given token sequence is recognized. The modifications to the grammar were directly applied within the "*cs-parser.jay*" file. During the parsing process an abstract syntax tree which consists of abstract representations of expressions and statements is generated.

There are many different possibilities of how a certain modification can be implemented. Instead of emitting code manually, the presented language features are based on modifications directly applied to the abstract syntax tree. This can be explained considering the following example. If an assignment statement is detected, an instance of the class `Assign` is created. This object contains a left-hand and a right-hand expression which are the target and the source of the assignment operation respectively.

During semantic analysis this object is resolved by identifying the types of the expressions provided and code generation can emit the appropriate code. In case of the timed assignment, the implementation could be similar and a new object could have been created which then emits the required code. Instead, the representation of the operation within the abstract syntax tree was modified to contain not only an assignment object but further objects representing method invocations to the run-time available QoS management framework. In other words, the current implementation reuses existing elements of the abstract syntax tree to provide the new functionality.

This implementation technique has the advantage to not worry about the Common Intermediate Language (CIL). Correct code generation is left to the existing implementation of the compiler. This approach allows easy modifications to the front-end of the compiler without worrying about the code generation in detail.

The use of timed data types as overlay types - this means that the additional type information is ignored when applying type compatibility checks - allows the generated CIL code to be standard conform. It further enables compiled applications to run on different .NET implementations and in addition allows QoS-aware libraries to be reused within standard .NET or Mono programs.

If timed data types had been integrated into the Common Type System (CTS), other programming languages could use the new types. This has the disadvantage that a QoS-aware .NET runtime needs to be shipped with each application instead of reusing existing installations. However, this approach was

not chosen to provide compatibilty to existing implementations which might also increase acceptance.

7.1 Applied Modifications

The following files are modified to implement the presented language enhancements for *Quality Aware Programming*.

- **assign.cs** - This file contains all classes which are required to implement an assignment operation and the required object initializer. The MMC# modification is required for the constraint assignment when initializing a time variable with a new constraint.
- **cs-parser.jay** - This is the parser definition file which reads the input source code and provides the abstract syntax tree. It contains most of the modifications which have been applied.
- **cs-tokenizer.cs** - The tokenizer is extended to accept new keywords and the timed assignment operation.
- **delegate.cs** - This file contains all classes which deal with delegates. The implementation of timed parameters requires modification to this file.
- **ecore.cs** - This file contains base classes for expressions and types. It includes an implementation of the timed data type which is implemented as an overlay datatype which can be applied to any other data type.
- **expression.cs** - All expressions are implemented within this file. Time parameter arguments are implemented as well as the cast operation which is required to ignore the timed type to provide correct type compatibility with non-timed types.
- **parameter.cs** - This file is responsible for method parameters. It contains the extensions for timed method parameters.

8 Conclusion and Future Work

This paper presents an approach of adaptive QoS-aware programming with the help of MMC#. It integrates a few novel language extensions directly within the common general purpose programming language C#. Furhtermore, we show how semantic analysis can be applied partly at compile-time and partly at runtime with the help of compiler-generated code. This analysis provides correctness checks and optimizes constraint definitions. Last but not least, the integration within the Mono framework is explained.

There are still many open questions which require further research. We need to define a set of programming patterns to show how the presented programming features should be applied in QoS-aware applications. Furthermore, a detailed analysis of system influence on QoS monitoring needs to be done. The impact of operating system scheduling and the use of garbage collection on QoS monitoring needs to be reflected in detail.

Furthermore, we want to check possibilties if compile time type safety can be extended to handle QoS negotiation within a QoS-aware multimedia framework.

All in all, this is one step to make QoS-aware programming widely available and to increase the number of adaptive applications by making their development much easier.

References

[1] C# Language Specification Standard ECMA-334, June 2005

[2] Common Language Infrastructure (CLI) Standard ECMA-335, June 2006

[3] , Nguyen,, Thi Viet Nga and Irigoin,, François: Efficient and effective array bound checking ACM Trans. Program. Lang. Syst., 2005, Vol 27, pages 527-570, New York, NY, USA

[4] Java, 2009 http://java.sun.com/

[5] Mono: Open Source .NET Development Framework, 2009 http://www.mono-project.com

[6] Mono (software), 2009 From Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Mono_(software)

[7] Miguel de Icaza: The Internals of the Mono C# Compiler, 2009 From Mono-Project Subversion

[8] Miguel de Icaza: The Mono C# Compiler, 2002 From Mono-Project, http://primates.ximian.com/ miguel/slides-europe-nov-2002/Mono_C_Sharp_Overview_1007.sxi

[9] List of CLI languages, 2009 From Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Microsoft_.NET_Languages

[10] Brian W. Kernighan and Dennis Ritchie: The C Programming Language, Second Edition Prentice-Hall, 1988, ISBN 0-13-110370-9