Mapping the IEC 1131-3 Language to the ECMA-335 Common Language Infrastructure and the Real-Time Operating System QNX

Holger Arnold* and Uwe Petermann†

Leipzig University of Applied Sciences
Department of Computer Science
Postfach 30 00 66, Leipzig 04251, Germany

March 2003, revised February 2005

---

1 Introduction

In this report we summarize the results and experiences of the *CommonCtrl* project. Its aim was to integrate a process control system based on the PLC programming language IEC 1131-3 with a dynamic object-oriented environment based on the ECMA-335 Common Language Infrastructure running on the real-time operating system QNX. This project was motivated by the demand to combine the standardized programming language IEC 1131-3, which is familiar to many engineers and which has proved to be a solid foundation for process control systems, with a platform that provides a large class library, dynamically loadable components, and interoperability to popular operating systems and communication standards. The present paper gives only an overview of the results, more details can be found in the full research report [1].

1.1 The IEC 1131-3 Language

The IEC 1131-3 standard [6] defines several textual and graphical languages for programmable controllers. For our project, a variant of the language IEC 1131-3 Structured Text was chosen as the source language, a static procedural language whose notation and abstraction level are similar to those of the Pascal line of programming languages. Its design follows the requirements of control environments: the memory requirements of a program can be calculated statically before execution and there are no non-deterministic operations like dynamic memory allocation. In practice, however, implementations commonly have to give up some of these properties due to restrictions of the underlying platform or to realize certain features.

The type system of IEC 1131-3 provides the following elementary data types: signed and unsigned integers of different lengths, bit sequences of different lengths, floating point types, booleans, strings, and date and time types. Based on them, users can define simple types, subrange types, enumeration types, array types, and structure types. Sequences of computation can be structured in functions, function blocks, and programs. These units are called program organization units. The task structure of a project and its runtime parameters are defined in a configuration.

*See `http://harnold.org/` for contact information.
†Contact: `uwe@imn.htwk-leipzig.de`

1.2 The Common Language Infrastructure

The Common Language Infrastructure (CLI) is an integrated platform for the development of component-based, object-oriented software. In many aspects similar to the Java platform [5, 7], it is based on a unified type system, the Common Type System (CTS), and a virtual machine which executes a type-safe Common Intermediate Language (CIL). Additionally, in the Common Language Specification (CLS), the CLI specifies rules for the interoperability of components written in different programming languages. Originally created by Microsoft, the CLI specification [4] has been submitted to the ECMA and has been accepted as the ECMA-335 standard.

1.3 The Runtime Environment

Our runtime environment consists of an implementation of the IEC 1131-3 standard, an implementation of the Common Language Infrastructure, and the underlying operating system. With respect to the IEC 1131-3 standard, our runtime environment is a software controller implementation running under the real-time operating system QNX. This environment has some special features which distinguish it from other implementations. The most notable of these features is the Real-time Data Base (RTDB), where all process variables of a project are stored. The process variables in a data base can be shared among several program organization units. Because the system automatically handles issues like transactions and synchronized access, this data base is a powerful mechanism for inter-program communication. Another important feature of the environment is the interactive development environment. The complete development, configuration, and control cycle of a CommonCtrl installation is done through this visual interface.

As an implementation of the Common Language Infrastructure, we use the *Mono* runtime, an open-source CLI implementation created by the Mono project (see *www.mono-project.com*). It is the most complete among the freely available implementations and can be embedded as a library into unmanaged C programs. This is used to enhance existing components, for example the Real-time Data Base or the visualization component, by the capability of executing CLI components. The Mono library also offers the functionality to access the metadata stored in Portable Executable (PE) files and in the runtime structures of Mono. The Mono runtime is being ported to the QNX operating system and modified where necessary to match our special requirements.

Because of the special properties of our runtime environment, our source language differs in several aspects from the IEC 1131-3 standard as defined in [6]; section 1.5 describes these differences.

1.4 Mapping Strategy

For the mapping of the IEC 1131-3 language to the Common Language Infrastructure, several strategies are reasonable, depending on the intended completeness and other requirements on the mapping. For our purpose, the requirements were:

1. IEC 1131-3 modules should be compilable to CLI modules that can be executed in any CLI compatible runtime environment, preserving the semantics of the IEC 1131-3 language and execution environment.

2. It should be possible to extract the IEC 1131-3 interfaces from such compiled modules, which means that the mapping from IEC 1131-3 to the CLI must preserve enough information to allow a reverse-mapping of the interfaces.

The first requirement is obvious; the second requirement ensures that IEC 1131-3 modules can be separately compiled and linked together. To make a reverse-mapping from the CLI to IEC 1131-3 possible, the semantics of the IEC 1131-3 interfaces have to be embedded into the compiled CLI modules. Of course, there is more than one way to do this, and the one we have chosen for several reasons can be described as follows:

1. The IEC 1131-3 constructs are mapped to their semantically equivalent counterparts in the CLI. We call the CLI types that are created as counterparts of IEC 1131-3 types *user types*. Because there are different IEC 1131-3 constructs that are mapped to the same CLI construct, this mapping is not sufficient to make the mapping of the IEC 1131-3 interfaces to the CLI invertible.

2. Information about the IEC 1131-3 interfaces is embedded into the CLI modules through special *meta types*. Note that these meta types are not related to the CLI metadata, but are normal CLI types which have a special meaning only in the context of IEC 1131-3. This is necessary because the CLI metadata is not able to capture all of the IEC 1131-3 type semantics.

In this article, we shall not describe the details of the meta types. To get an understanding, it is sufficient to know that a meta type contains things like the name of the base type (in the `BaseType` field), the kind of the type (in the `TypeKind` field), and the value that instances of the type are initialized with (in the `InitialValue` field).

1.5 Source Language

Because of the properties of our runtime environment and because of further implementation constraints, there are a number of differences between our source language and the IEC 1131-3 standard as it is defined in [6], in particular:

- All inter-program and inter-process communication is done using process variables stored in the Real-time Data Base component. This feature supersedes global variables, external variables, and access paths.

- Because we have a pure software implementation, there are no directly (hardware) addressed variables.

- Currently, there are no edge variables (i.e. the types `BOOL R_EDGE` and `BOOL F_EDGE`). They are superseded by a corresponding type `PULSE_PV`.

- Only complete programs, not single function blocks, may be assigned to a task. The reason for this is that the semantics of assigning function blocks to tasks are not clear from the IEC 1131-3 specification.

- Because the configuration of a CommonCtrl system is normally done through the interactive development environment, the `CONFIGURATION` section as specified in the IEC 1131-3 standard is not used. A different configuration format, adapted to the specific requirements of our implementation, will be specified in the future.

- In addition to the standard declaration of input, in-out and output variables of a function block, which have call-by-value semantics, the modifier `REF` is provided to declare such variables to have call-by-reference semantics.

2 Data Types

We do now describe how the IEC 1131-3 language is mapped to the Common Language Infrastructure. We begin with the mapping of data type declarations.

2.1 Data Type Declarations

User-defined data types can be declared in IEC 1131-3 as named or as anonymous types. A data type declaration can either construct a new subrange, enumeration, array, or structure type, or derive a type from an already existing type, its direct base type. A derived type only defines a new type name and possibly a new initial value, but not a structurally different type. The transitive closure of the direct base types of a type T forms the set of base types of T. Derived types are assignment compatible to all their base types, which means that values of derived types can be assigned to variables of their base types.

If a type is directly derived from another type, it is mapped to the same type as its direct base type and no new user type is created. However, a new meta type is created for every named type that is declared.

2.2 Anonymous Types

Anonymous data types, which can be enumeration, subrange, and array types, are declared within variable declarations, which are described in section 3.1. For every anonymous type the IEC 1131-3 compiler has to generate a unique identifier.

Anonymous types are more general than equally defined named types. More specifically, every named type T' which is equally defined to an anonymous type T can be considered a subtype of T. The following rules describe what "equally defined" means in this context:

(a) Subrange type declarations are equal if they have the same base type. The minimum and maximum values are not considered, which is consistent with the IEC 1131-3 rule that a subrange type is a subtype of its base type (which in general has different minimum and maximum values).

(b) Enumeration type declarations are equal if they define the same set of named constants in the same order.

(c) Array type declarations are equal if they have the same element type, the same number of dimensions, and, for each dimension, the same lower and upper bounds.

All equal anonymous type declarations refer to the same anonymous type. This means that whenever the IEC 1131-3 compiler processes an anonymous type declaration, it must check whether an equal anonymous type has already been defined.

Although a named type is a subtype of the equally defined anonymous type, this relation is not reflected in its meta type, because this would mean that the declaration of a named type implicitly declared its anonymous supertype, even if the anonymous type is not used. Instead, anonymous types are marked as such in their meta types, and the IEC 1131-3 compiler has to ensure that a value of a named type can be assigned to a variable of an equally defined anonymous type. On the side of the CLI, the assignment compatibility between anonymous types and equally defined named types is guaranteed because

(a) equally defined subrange types are mapped to the same elementary CLI type (see section 2.6),

IEC 1131-3 Type	Corresponding CLI Type
SINT	int8
INT	int16
DINT	int32
LINT	int64
USINT	unsigned int8
UINT	unsigned int16
UDINT	unsigned int32
ULINT	unsigned int64
REAL	float32
LREAL	float64
TIME	System.TimeSpan / int64
TIME_OF_DAY	System.DateTime / int64
DATE	System.DateTime / int64
DATE_AND_TIME	System.DateTime / int64
STRING	string
BOOL	bool
BYTE	unsigned int8
WORD	unsigned int16
DWORD	unsigned int32
LWORD	unsigned int64

Table 1: Elementary data types of IEC 1131-3 and corresponding types in the Common Language Runtime

(b) equally-defined enumeration types are compatible because all CLI enumeration types are fully interchangeable with their underlying type (which is always `int32` in our case; see section 2.7), and

(c) equally-defined array types are mapped to the same CLI array type (see section 2.8).

2.3 Elementary Data Types

For every elementary IEC 1131-3 type there is a corresponding elementary or library type in the Common Language Infrastructure. Table 1 shows the elementary data types of IEC 1131-3 and their corresponding CLI types. Variables of elementary types are mapped directly to variables of their corresponding types. Because the elementary types are fixed and known to the IEC 1131-3 compiler, no additional meta type information is required for them.

The string type is the only type whose semantics differ between IEC 1131-3 and the CLI; consequently, it needs a special handling. The problem is that in IEC 1131-3, strings are mutable value types, but in the CLI, they are immutable reference types. The solution is to embed a native CLI string into a value type whose only field is a public string reference. In this way, strings can be treated as value types, and modifications of a string can be implemented by replacing the string reference. When we speak of CLI strings in this text, we refer to the value type embedding the actual string reference.

The date and time types differ from the other elementary types in that they have two corresponding CLI types. To be compatible with the CLI library and to profit from their predefined functionality, they are mapped to the library types `System.TimeSpan` and `System.DateTime`. But if time or date values are to be stored in the metadata as initial values, their `int64` equivalent

is used. (`System.TimeSpan` and `System.DateTime` are internally represented as `int64` values. Both have constructors that take a single `int64` value and a `Ticks` property to read the `int64` value.)

2.4 Generic Types

As the generic types of IEC 1131-3 are only used as argument types in predefined program organization units and are not available for user definitions, no additional type information is required for them. Calls to predefined functions and function blocks with generic argument types must be resolved by the IEC 1131-3 compiler to calls to equivalent functions and function blocks with non-generic argument types, as described in the IEC 1131-3 standard [6]. For example, a call to the predefined generic function `ADD` using `REAL` arguments would be resolved to a call to the predefined non-generic function `ADD_REAL`.

2.5 Simple Types

A simple type is a type that is directly or indirectly derived from an elementary type. Simple types are mapped to the same CLI types as their base types (the elementary or library types listed in table 1); only a new meta type has to be defined for a simple type.

2.6 Subrange Types

Subrange types are integral types that are limited to a subrange of their base types. The IEC 1131-3 standard does not define how the execution environment should react when a value of a subrange type exceeds its range. If the value of a subrange type is to be supervised, it is the responsibility of the IEC 1131-3 compiler to insert appropriate instructions to check the value of a subrange type. A CLI conform approach would be to raise an exception when the value gets out of range.

Subrange types are mapped to the same elementary types as their base types, so that only a meta type has to be defined for a subrange type. The valid range of a subrange type is stored in two constants, `MinValue` and `MaxValue`, in the meta type.

2.7 Enumeration Types

Enumeration types are sets of named constants. Their numeric values are not visible from IEC 1131-3. For every IEC 1131-3 enumeration type that defines a new set of constants (one that is not derived from another enumeration), a new value type extending `System.Enum` is defined. The constants are mapped to distinct `int32` values; the constants and their values are stored in literal static fields of the enumeration type. The definition of the enumeration type must follow the rules in the CLI specification, Part. II.

If an IEC 1131-3 enumeration type is derived from another enumeration type, the derived type is mapped to the same type as its base type, and no new CLI enumeration type needs to be created. However, for every enumeration type, whether it defines a new set of constants or is derived from another enumeration, a new meta type is required.

2.8 Array Types

Array types in IEC 1131-3 can be single-dimensional or multi-dimensional. For every IEC 1131-3 array type there is a corresponding array type in the CLI. This is the CLI array type

(i) whose element type is the mapped element type of the IEC 1131-3 array type, and

(ii) that has the same rank (number of dimensions) as the IEC 1131-3 array type.

Because in the CLI the bounds of an array are not part of its type, multiple IEC 1131-3 array types can correspond to one CLI array type. In the CLI, array types are not explicitly created by the user, but are automatically provided by the runtime as subtypes of `System.Array`. Thus, only a meta type has to be created for an array type.

As array types are reference types in the CLI, there has to be a method to create copies of an array, which is called when an array is used as a by-value parameter to a program organization unit. Because the elements of an array can be reference types as well, a deep copy is required, and the `Clone()` and `Copy()` methods provided by the `System.Array` class cannot be used for this purpose. Therefore, for every array type, a static method `DeepCopy()` must be defined in its meta type, which creates a deep copy of an array of this type. The signature of this method for an array type T is

`public static final hidebysig` T `DeepCopy(`T a`) cil managed`

It should implement the following pseudo-code:

- Create a new array b of type T with the same lower and upper bounds as a. The elements of the array are zero-initialized by the runtime.
- For every element index i of T (i is an n-tuple, where n is the number of dimensions of T):
  - Let a_i be the i-th element of a and b_i the i-th element of b. Let E_T be the element type of T.
  - If E_T is a value type, set b_i to the value of a_i. If E_T is a reference type, i.e. an array or structure type, set b_i to the result of `IEC1131.Metatypes.`E_T`.DeepCopy(`a_i`)`.
- Return b.

As noted above, the bounds of a CLI array are not defined by its type, but specified in the constructor call when an array is created. Because the bounds of an array are required to compute correct indexes, they must be stored in the array's meta type. Additionally, the IEC 1131-3 name of the element type is stored in the meta type.

2.9 Structure Types

Structure types group several named elements of different types. They are mapped to CLI classes that extend `System.Object`, and their elements are mapped to public fields whose types are the mapped types of the element types.

Because structure types are mapped to reference types in the CLI, there has to be a method to create copies of a structure, which is called when a structure is used as a by-value parameter to a program organization unit. Because the elements of a structure can be reference types as well, a deep copy is required, and the `MemberwiseClone()` method provided by the `System.Object` class cannot be used for this purpose. Therefore, for every structure type, a static method `DeepCopy()` must be defined in its meta type, which creates an deep copy of a structure of this type. The signature of the method for a structure type T is

`public static final hidebysig` T `DeepCopy(`T a`) cil managed`

It should implement the following pseudo-code:

- Create a new object b of type T. The elements of the object are zero-initialized by the runtime.
- For every element x_i of T:
  - Let T_i be the type of the element x_i.
  - If T_i is a value type, set $b.x_i$ to the value of $a.x_i$. If T_i is a reference type, i.e. an array or structure type, set $b.x_i$ to the result of `IEC1131.Metatypes.`T_i`.DeepCopy`$(a.x_i)$.
- return b.

A structure type that is derived from another structure type is mapped to the same class as its base type. For both cases, new and derived structure type, a new meta type is created. In addition to the standard meta information, the meta type of a structure type has to store the IEC 1131-3 types of the structure elements.

3 Program Organization Units

In the sections above, we have described how data type declarations are mapped from IEC 1131-3 to the CLI. We continue with the mapping of the program organization units, which are sequential subprograms used to structure an IEC 1131-3 project.

3.1 Variable Declarations

Variables can be declared in IEC 1131-3 only in the context of a program organization unit. Depending on the type of the enclosing program organization unit, a variable declaration is mapped to different CLI constructs. The type of a variable can be any named type that has been declared before the variable declaration. Additionally, IEC 1131-3 allows the anonymous declaration of subrange, enumeration, and array types within a variable declaration, as described in section 2.2.

3.2 Functions

A function defines a mapping from a set of input values to a single output value. The result of a function must depend only on its input parameters. Functions are mapped to methods in the CLI. While the CLI allows the definition of global methods (to be precise, such methods are not really global, but are assigned to an internally generated type `<Module>` and are said to be defined at module scope), they cannot be referenced from outside the assembly they are declared in. For this reason, a function is mapped to a static method `Invoke()` that is embedded in a class whose name is equal to the name of the function.

In addition to the class containing the `Invoke()` method, a meta type is created for every function. The meta type identifies the type as a function and provides the inverse mapping of the function's parameter and return types. The name of the meta type is equal to the name of the function (but in a different namespace). The signature of the function, return type plus parameter types and names, is encoded in a string constant.

Input Declarations

The input declarations of a function are mapped to by-value input parameters and therefore appear in the signature of the `Invoke()` method. For arrays and structures, an explicit copy to a

local variable is created through a call to the `DeepCopy()` method defined in the corresponding meta type. The local copy is then used as the input parameter.

Variable Declarations

Variable declarations are mapped to local variables of the `Invoke()` method. For value types, the `locals init` declaration (which is reflected in the CLI metadata) is sufficient; arrays and structures have to be created using the `newobj` instruction or the `newarr` instruction in the case of single-dimensional zero-based arrays.

All local variables have to be initialized, either to the initial values given in the variable declaration, or, if no initial values are given, to the initial values of their types. The initial values of value types are elementary values that are simply stored in the corresponding local variables using sequences of `ldc.*` and `stloc.*` instructions. Strings are initialized using the `ldstr` instruction, where the initial strings are stored in the #US heap. For arrays and structures, the IEC 1131-3 compiler has to derive the appropriate initialization instructions from the encoded initial values stored in their meta types.

Constant Declarations

References to constant declarations of value types are directly inserted into the generated CIL code. They are loaded onto the stack using the `ldc.*` and `ldstr` instructions. Constant declarations of reference types are mapped to private static init-only fields that are created and initialized by a type constructor. The types of these fields are the mapped types of the constants. The initialization is the same as for variable declarations, except that references are made to static fields instead of local variables.

3.3 Function Blocks

The main difference between function blocks and functions is that instances can be created of function blocks and that these instances can have local data that is preserved between invocations. Function blocks are mapped to classes that inherit from `System.Object` in the CLI, which makes them reference types. The input and output declarations of a function block are mapped to public instance fields. The statements of a function block are embedded in a parameter-less instance method `Invoke()`, and the invocation of a function block is mapped to a call to this method.

In addition to the function block type, a meta type is created for every function block. The meta type stores the full signature of the function block, which contains the names and types of all input and output variables. The signature is stored in separate string constants for input, output, and in-out variables.

Declarations of local variables and constants in function blocks are handled in exactly the same way as those in functions. What differs is the mapping of input and output declarations, since they have different semantics in functions and function blocks.

Input and Output Declarations

The input and output declarations of a function block, declared with `VAR_INPUT`, `VAR_IN_OUT`, and `VAR_OUTPUT`, are mapped to public instance fields. On the side of the CLI there is no difference between an input, output, or in-out declaration. The IEC 1131-3 compiler must ensure that the semantics of IEC 1131-3 (for example, output variables can only be written within the function block body) are respected.

Input and output variables are initialized in a parameter-less instance constructor, which is automatically called when function block instances are created using the `newobj` instruction. The initial values are stored in the meta type of the function block, and the IEC 1131-3 compiler has to derive appropriate initialization instructions from these initial values.

Read and write operations on input and output variables of a function block instance are mapped to read and write operations on the corresponding fields of the function block object. The `ldfld` and `stfld` instructions are used for this purpose. Because structures and arrays are mapped to reference types, to copy the value of such a type between a field and a variable or another field, it must be explicitly copied using the `DeepCopy()` method of the corresponding meta type.

Input and Output Reference Declarations

In practice, function blocks are often designed to operate on larger structures. Such structures are passed to the function blocks as input or in-out variables, or are returned from the function blocks as output variables. With the mapping described in the previous section, such variables are copied when they cross function block boundaries, which means that in-out variables are even copied twice. With larger structures, this copying can cause a considerable overhead.

As an alternative, input and output variables can be mapped to call-by-reference parameters of the `Invoke()` method. This has the following consequences:

1. Literals and constants cannot be assigned to reference variables because they cannot be modified.

2. Variables that are assigned to reference variables outside a function block can be modified directly from within the function block and vice versa.

3. All reference parameters must be supplied in a function block invocation, the values of these variables are not retained between successive invocations, and they cannot have default values.

It is not clear whether these points are in conformance or in violation with the IEC 1131-3 standard, because the intended behavior is not exactly specified. Because of this, it seemed reasonable to us to introduce a new keyword to declare input and output variables as reference parameters. We assume here that such variables are declared using the modifier "`REF`".

For input and output references, variables of value types are passed using managed pointers of their types. For variables of reference types, namely arrays and structures, their references are passed directly. The difference to non-reference input and output variables is that they are no longer copied using their `DeepCopy()` method.

Inside the function block, read and write operations on reference variables of value types are mapped to read and write operations on the values referenced by the corresponding method parameters. The `ldind.*` and `stind.*` instructions are used for this purpose, after loading one of the arguments on the stack using one of the `ldarg.*` instructions. Read and write operations on reference variables of reference types are mapped to read and write operations or method calls on the object references passed as parameters.

3.4 Programs

Programs are essentially top-level function blocks, at least in the subset of the IEC 1131-3 language we consider. Programs are mapped in the same way as function blocks. This includes input and output variable declarations, and the meta type.

4 Runtime Aspects

In several aspects, the properties of a dynamic managed runtime environment, like the Common Language Infrastructure or the Java platform, are in contrast to the runtime requirements of an embedded system under real-time conditions. This section discusses some approaches how these properties can be reconciled with the real-time requirements.

4.1 Tasks

In our setting, the mapping of the configuration of an IEC 1131-3 project reduces to the question how IEC 1131-3 tasks are mapped to executable entities of the underlying operating system, including the specification of scheduling priorities and execution intervals.

We have decided to map IEC 1131-3 tasks to operating system processes. This means that every IEC 1131-3 program runs in its own process that executes an instance of the CommonCtrl runtime environment (a modified Mono runtime). Programs, function blocks, functions, and data types are stored in CLI assemblies and are loaded by the CommonCtrl runtime on demand. The loading can be triggered either by the runtime, when a currently unloaded assembly is referenced, or by the user of the system, through one of its control interfaces.

What is important is that each running IEC 1131-3 program has its own runtime environment, which is executed in a separate operating system process. Compared to separating multiple IEC 1131-3 programs only by threads running under a shared runtime environment, this has the advantage that IEC 1131-3 programs are more isolated from each other. Consequently, the interaction between a program and its runtime environment does not influence the execution of the other programs beyond what is inevitable on any system with limited resources. This especially means that a task of higher priority is never interrupted by a task of lower priority, even if the lower priority task performs system-level operations like memory management. As the Mono runtime can easily be compiled into a shared library, executing a separate runtime environment for every IEC 1131-3 program should not lead to any significant memory overhead.

4.2 Automatic Memory Management

Despite all attempts to create real-time garbage collectors, automatic memory management, or garbage collection, is still a major obstacle to the real-time behavior of systems. The reason is that garbage collectors modify the memory they manage and therefore cannot be safely interrupted by other threads in the same process. Usually, garbage collectors must run uninterrupted until they explicitly return control.

Mono's Current Memory Manager

The Mono runtime currently uses the conservative garbage collector by Hans Boehm, which is described in [2, 3]. The Boehm collector has been developed as a replacement for the `malloc` function in C or the `new` operator in C++. As C/C++ runtime environments provide no explicit object layout information, the collector must scan the memory for potential object references, building up its own dependency graph. As the dependencies are determined by a heuristic search for memory locations containing pointers, there is no guarantee that the resulting dependency graph is complete. In an environment like the CLI, where all the necessary type and object information is available, using such a "blind" garbage collector is not an ideal solution. Therefore, the Boehm garbage collector can also be supplied with explicit layout information.

In its default mode, the garbage collector stops all user threads when an allocation request fails because there is not enough free space left. It then scans all memory blocks and releases

those that are unreachable. During its operation the garbage collector must not be interrupted by other threads in the process. The collector always runs in the thread that triggered the collection by a memory allocation request; there is no separate garbage collector thread.

Additionally, the Boehm Garbage Collector can be used in an incremental mode, in which it scans only those blocks that have been modified since the last run. This does not guarantee a time-bounded operation of the garbage collector, but it usually makes single runs significantly faster. This feature requires some support from the operating system and the underlying hardware platform to detect modified memory pages, because a write barrier must be placed upon the supervised pages. A write access to such a page then causes a protection fault which is caught by the garbage collector. The downside of this approach is that it leads to problems with operating system calls writing to such protected memory pages, as system calls cannot be interrupted by the memory manager. Working around this problem is possible, but expensive. Therefore, Mono is not compatible with the incremental mode.

In our view, memory management is the real Achilles heel of the Mono environment and causes severe problems not only for our project. The unbounded operation of a garbage collector is problematic for any system under real-time conditions. Even more problematic, however, is that the current Mono runtime is not very good a reclaiming unused memory. It is easy to write programs whose unreclaimed memory grows monotonic with the running time of the program. This behavior makes Mono currently unusable for long-running programs, especially in an automation environment, where reliability is one of the most important factors.

4.2.1 Mono's Future Memory Manager

The bad performance of Mono's memory management has several reasons. First, being a garbage collector for C and C++, the Boehm collector is inherently incomplete, as described above. This is problematic because it is used not only for CLI objects (for which layout information is available), but also for memory allocated in the runtime for C objects (for which no layout information is available). Second, for the same reason of being a C/C++ memory manager, the Boehm collector cannot compact reclaimed memory, as this would mean to move active objects around. Depending on the application, this can cause serious memory fragmentation. Third, there are some technical details leading to non-optimal behavior of the memory manager.

Although some significant improvements in memory management have already been achieved by the Mono developers, building a reliable and efficient memory manager into Mono eventually means to replace the Boehm garbage collector. Besides creating or adapting an efficient garbage collector, which is already a complex task on its own, this also requires a complete re-design of Mono's internal memory management. Because of the importance of memory management for Mono's success, this feature is actually targeted for version 2.0 of Mono.

Memory Management in the Context of IEC 1131-3

As the IEC 1131-3 language provides no mechanism for explicit dynamic memory allocation, the memory requirements of an IEC 1131-3 program are statically known before the program is run. The mapping to the CLI, however, does not preserve this property: instances of arrays, structures, in some cases strings, and function blocks are created dynamically. In contrast to other implementations of IEC 1131-3, such objects cannot be allocated on the stack when mapped to the CLI. This can cause a continuous heap growth during the run time of a program and makes a mechanism for storage reclamation absolutely necessary.

For the CommonCtrl system, we see no way to fully circumvent the problems of automatic memory management; the system has to live with the properties of Mono and its garbage collector, being it the Boehm collector or a different one in the future. As a consequence, the CommonCtrl

system can only be used as a soft real-time system, but this is also a property of the existing IEC 1131-3 runtime environment.

4.3 Just-in-Time Compilation

The Mono runtime environment performs just-in-time compilation. This means that a method is not compiled to native machine code until it is called the first time. This avoids the compilation of methods which never get actually called, which can be most of the methods in a larger library. However, this feature can also have the effect that the execution of a method containing call instructions triggers the compilation of a number of other methods. The compilation of a large chain of methods can cause a noticeable delay.

A simple solution to this problem is to ensure that, before a method gets executed, this method and all methods possibly called by this method, either directly or indirectly, are compiled. This is achieved by a small addition to the Mono runtime.

References

[1] Holger Arnold and Uwe Petermann. Mapping IEC 1131-3 to the ECMA-335 Common Language Infrastructure and the Real-Time Operating System QNX. Research report, January 2003.

[2] Hans J. Boehm. Space efficient conservative garbage collection. *ACM SIGPLAN Notices*, 28(6):197–206, 1993.

[3] Hans J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, 18(9):807–820, 1988.

[4] ECMA International. *Standard ECMA-335: Common Language Infrastructure*, December 2001. Available at `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[6] International Electronic Commission. *Programmable controllers, Part 3: Programming languages (Standard IEC 1131-3)*, August 1993.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.