Requirements for a Real-Time .NET Framework

A. Zerzelidis and A.J. Wellings
{alex, andy}@cs.york.ac.uk
Department of Computer Science, University of York, U.K.

Abstract: *The Microsoft .NET Framework is a comparatively new technology that has already gained considerable momentum. Its user base and popularity is expanding. In addition, it offers a number of important traits, such as code portability and cross-language development. It is these features that have attracted our attention into investigating the possibility of using .NET for architecture -neutral real-time systems. As a result, this paper tries to set the groundwork for implementing a real-time version of the Microsoft .NET Framework by specifying a set of requirements.*

1 Introduction

With embedded computers becoming more and more omnipresent, and with the world becoming more and more interconnected, there is a need for architecture-neutral real-time systems (*a real-time system whose target architecture is unknown at system design time)*. Also, interconnection increases the possibility that systems developed by different vendors might need to interoperate. And since different vendors could possibly use different implementation languages, there is a need for multi-language working for real-time systems development. The Microsoft .NET Framework offers both an architecture-neutral platform and a multi-language development environment. It is our aim to examine how feasible it is to introduce real-timeliness into .NET.

So far there have been few attempts to examine the .NET Framework from a real-time perspective. The two mentioned below examine the .NET Compact Framework in conjunction with Windows CE. The first one is an attempt to examine .NET's

suitability for real-time systems [Lutz03]. The article looks into what .NET has to offer with respect to various real-time features like scheduling, memory management, and physical memory access. It concludes that .NET and C# are not suitable for hard real-time systems. [Struys03] is a second article that examines the real-time behaviour of .NET. However, this article does not consider .NET as a platform for carrying out real-time operations, but rather as a presentation layer to a real-time Win32 application running on a real-time operating system (Windows CE).

In contrast with the above work, the overall goal of our research is to produce a real-time .NET Framework (RT.NET[1]). As a first step towards this goal, this paper generates an initial set of real-time requirements for the .NET Framework. Rather than starting from the beginning, though, we take as a starting point the requirements presented in the report for Real-Time Java, produced by the National Institute of Science and Technology (NIST) [NIST99].

1.1 Introduction to the .NET Framework

The .NET Framework is a development platform. As such it consists of a runtime environment, called the Common Language Runtime (CLR), on which programs developed for .NET run, and a set of types (classes) in the form of libraries. There are two main libraries required for a minimum implementation of .NET: the Base Class Library (BCL), which provides a simple runtime library for modern programming languages, and the Runtime Infrastructure Library, which provides the services needed by a compiler to target the CLI and the facilities needed to dynamically load types from a stream in the file format specified.

By introducing .NET, Microsoft's main aim was to provide a new development platform more capable of leveraging technologies than previous Microsoft platforms [Richter02], and indeed more capable than third-party platforms, e.g. Java. .NET's design goals include the following [Richter02]:

- simple and consistent, object-oriented programming model
- platform independence

[1] By "RT.NET" we shall refer to the real-time augmented .NET Framework and all its functionality.

- programming language integration
- automatic memory management
- type-safe verification
- security

To help the integration of languages into the .NET Framework, the CLR defines two sets of features: the Common Type System, or CTS for short, and the Common Language Specification, or CLS. The CLS is a true subset of the CTS. One can think of the two in terms of the following phrase: a language targeting the CLR cannot offer less than the CLS and cannot require more than the CTS. So the CTS is the superset of all language features offered by the CLR, and the CLS is the minimum set of features offered by a .NET language. Figure 1 demonstrates this [Richter02].

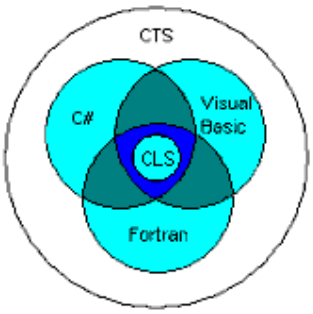

Figure 1: Relation between .NET languages, the CTS, and the CLS.

Code targeting .NET can be expected to either be stored and run locally, execute locally but get distributed over the Internet, or be executed remotely [MSDN]. Using .NET the developer can create the following types of applications [Richter02]: XML Web services, Web Forms, Win32 GUI applications, Win32 console applications, services, utilities, and stand-alone components. Primarily, though, .NET has been geared towards Internet applications and Web services. To get a better understanding of where .NET stands with respect to the Internet, [Richter02] provides the following diagram (Figure 2), illustrating an analogy between the role of an operating system and the .NET Framework. Here .NET is shown as the abstraction layer on top of the Internet, that helps/enables applications to take advantage of various Internet services, the same way as an operating system is the abstraction layer on top of the actual machine hardware, enabling the application to take advantage of the various peripherals.

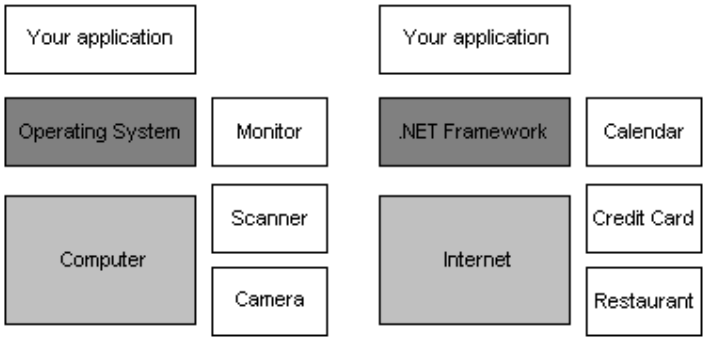

Figure 2: .NET – OS analogy.

.NET has also got its own programming language, C#, pronounced "C-sharp". C# is an object-oriented language. Although there are a number of languages developed by Microsoft that target .NET (Managed C++, Visual Basic .NET, Visual J# .NET, JScript), C# is specifically designed to help the programmer better leverage the capabilities of .NET. Only programming directly in the Common Intermediate Language (CIL, or IL for short), which is the .NET "assembly" language, can give the programmer complete control of .NET facilities.

From a real-time systems perspective we are not primarily interested in the Internet applications of .NET. Although the ease of interconnection .NET brings might be of interest to a distributed real-time audience, we are primarily focused on its virtual machine. More specifically, we are interested in introducing real-time facilities such as real-time scheduling (e.g. EDF or fixed-priority scheduling), thread abstractions (periodic, sporadic, aperiodic threads), and real-time clocks.

To allow the programmer to tap into these facilities, we will consider appropriate additions/alterations to C#. More specifically, we will consider how the language's current features can be used to support any of the real-time facilities. For example, C# has an event handling mechanism that could possibly be extended for asynchronous event handling. Moreover, drawing from the RTSJ, we could consider how the C# exception handling mechanism can be extended to cover asynchronous transfer of control.

1.2 Structure of the Report

Section 2 of this report will briefly present the NIST report on Real-Time Java. Section 3 will make a short introduction to the .NET Framework, what it is, its design goals, and the approach we're considering to modifying it for real-time. Section 4 is the main part of the report, dealing with the application of the NIST requirements to .NET. The section starts with the guiding principles, moving on to the concepts of a core functionality and extending profiles. Then come the actual requirements for .NET, followed by future goals for RT.NET, and rejected goals. Finally, section 5 presents our concluding points.

2 The NIST Report

The interest for extending Java to make it more appropriate for a wide range of real-time systems stemmed from Java's success. The NIST [NIST99] report aimed at coordinating such an effort, since much of the early work in this area was fragmented and lacked clear direction. The guiding principles the NIST Requirements Group set were that i) Real-Time Java (RTJ) might involve compromises between ease of use and efficiency, ii) that it should support the creation of software with extended lifetimes, and iii) that it should take into account current real-time practices and facilitate advances in the state of the art of real-time systems implementation technology. Of the three principles the last one is the most important. To achieve consistency with the current state of real-time practice, the following facilities were deemed necessary [NIST99]:

- Fixed-priority, round-robin scheduling.
- Mutual exclusion locking (avoiding priority inversion).
- Inter-thread synchronization (e.g. semaphores)
- User-defined interrupt handlers and device drivers — including the ability to manage interrupts (e.g., enabling and disabling).
- Timeouts and aborts on running threads.

The NIST group also suggested that profiles (subsets) of RTJ were necessary in order to cope with the wide variety of possible applications. According to this, real-time functionality for Java should not be provided as one big package, but should rather be split into a number of profiles, each one serving the needs of particular application domains. Underlying all these profiles, though, is the Core. By "Core" it is meant the basic real-time functionality found in any real-time operating system. As the NIST report puts it, "we can think of the Core as the intersection of all real-time capabilities provided by widely available real-time operating systems". All the various profiles either augment or restrict the Core.

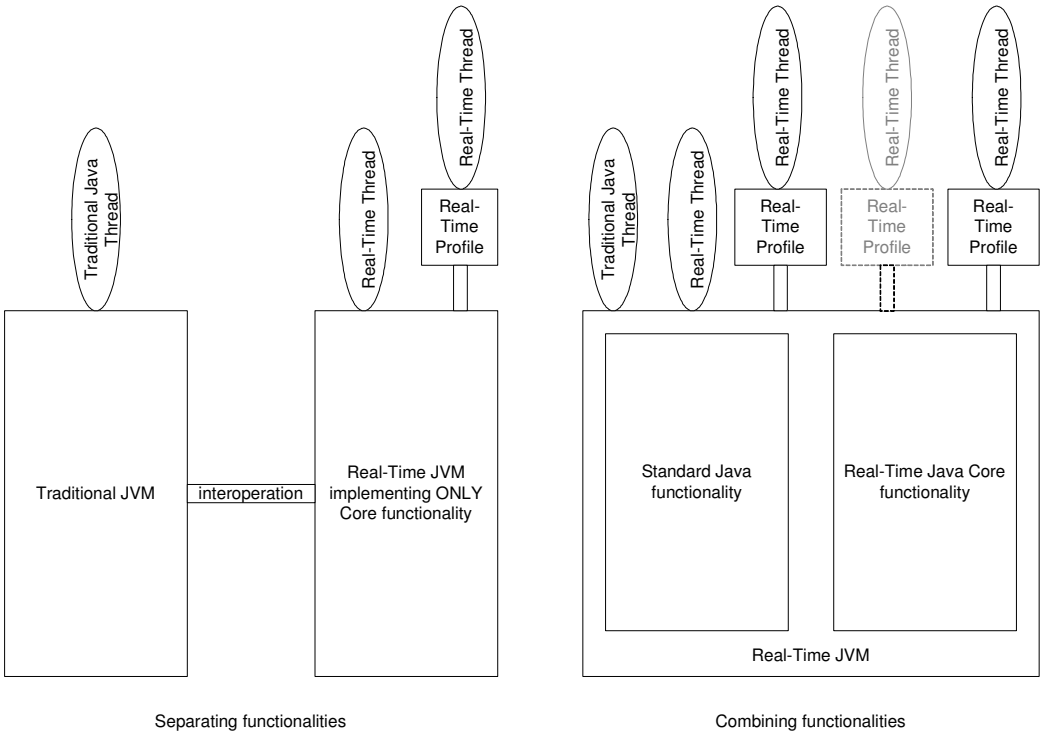

Figure 3: Two ways of implementing the Core and Profiles

Some examples of profiles are given below:

- Safety critical
- High availability and fault tolerant,
- Low latency,
- Deadline-based scheduling, or priority, or round-robin, or none
- Distributed real-time

There are two possible deployments that can be seen in Figure 3. The first is building a separate Real-Time JVM that can interoperate with any traditional JVM. In this case the profiles are only built on top of the Real-Time JVM. Traditional threads cannot run on the Real-Time JVM while real-time threads can *only* run on the Real-Time JVM. The two JVMs run as different processes on the same machine. In order to have an application running both traditional and real-time threads an interoperation mechanism has to be defined. This is the approach taken by the J-Consortium's Real-Time Core Extensions for Java (RTCE) [RTCE00]. RTCE was developed with the NIST report in mind and is totally compatible with that report.

The second possible deployment is to build a virtual machine that combines standard Java functionality with the real-time functionality offered by the Core. As can be seen in Figure 3, traditional Java threads can run on this virtual machine together with real-time threads. Profiles can be built on top of this JVM, and when these profiles are part of an existing deployment, real-time threads can use their additional functionality.

This second approach is partially taken by the Sun-backed Real-Time Specification for Java (RTSJ) [RTSJ00]. RTSJ does specify a new, real-time-enabled virtual machine, capable of running both traditional Java threads and real-time Java threads. However, it does not make use of the concept of Core functionality and profiles, trying, instead, to offer a complete set of real-time functionality in one package, and thus setting itself apart from the NIST report. Different real-time application domains are not individually targeted. Instead, RTSJ specifies high-level real-time constructs suitable to all real-time programming environments, trying to lift the burden from the real-time programmer of dealing with low-level programming details. However, recently attempts have been made to create subsets of the RTSJ suitable for high-integrity systems [Puschner2001, Kwon2002].

The main part of the NIST report, though, is an agreement that any implementation of RTJ should provide the following:

- A framework for finding available profiles.

- Bounded pre-emption latency on any garbage collection.
- A well-defined model for real-time Java threads.
- Communication and synchronization between real-time and non real-time threads.
- Mechanisms for handling internal and external asynchronous events.
- Asynchronous thread termination.
- Mutual exclusion without blocking.
- The ability to determine whether the running thread is real-time or non real-time.
- A well-defined relationship between real-time and non real-time threads.

3 Relationship between Java and .NET

The reason for taking the NIST report as a guideline is the close relation between C#/.NET and Java/Java Platform. Examining both technologies yields considerable commonalities between them ([NIST], [Cornelius02]). The most important are listed below. Moreover, both are targeting the same market (Web services and handheld devices amongst others).

- Both technologies promote portability through the use of a virtual machine (the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) respectively).
- Both technologies are highly dynamic, supporting object and thread creation at run-time.
- Both Java and C# are C++-derived object-oriented languages, supporting dynamic class loading and dynamic assembly loading respectively.
    - Both languages are organised into a class hierarchy with `System.Object` as the root class.
    - A class may implement more than one interface.
    - Method overriding is possible.
    - The programmer can control the visibility of classes and their members.

- Both technologies make use of automatic memory management (garbage collection).
- Both technologies support distributed applications.
- Both technologies are designed to support component integration and reuse.
- Both Java and C# provide well-defined execution semantics.
- Both Java and .NET aim at increased programmer productivity.
- Both technologies place value on security.

Apart from the similarities, though, .NET has some unique characteristics that must be taken into consideration ([Richter02], [Cornelius02]):

- .NET supports programming language integration, allowing the sharing of types between different languages.
- .NET has the delegate type, which is a type-safe pointer-to-function type. This means that the CLR always checks to see if the correct type of function is pointed at.
- In certain contexts, .NET performs automatic boxing and unboxing of values into the corresponding value types.
- Pointer types can be used in C#, although not in managed code (that is, not in code managed by the CLR). Code using pointers has to be declared `unsafe`. Any variables created in such code are not placed in the garbage collected heap.
- .NET, being developed by Microsoft, is, at the moment, best implemented on the Windows line of operating systems. Specifically, Windows XP and the Windows .NET Server Family have integrated support for .NET services, while Windows 2003 ships with the full .NET Framework.

4 Applying the NIST Requirements to .NET

Drawing from the NIST report, we can proceed to formulate the guiding principles, cross-disciplinary requirements, and future goals for real-time extensions to .NET.

4.1 Guiding Principles

The NIST guiding principles can be straightforwardly adopted for .NET and are as follows:

1. The design of RT.NET should allow compromises that improve ease of use at the cost of less than optimal efficiency or performance.
2. RT.NET should support the creation of software with useful lifetimes that span multiple decades, maybe even centuries.
3. RT.NET requirements are intended both to be pragmatic, by taking into account current real-time practice, and visionary, by providing a roadmap and direction to advance the state of the art.

Maintaining consistency with the state-of-practice entails the introduction in .NET of the same facilities as in Java, like fixed-priority round-robin scheduling, mutual exclusion locking with priority inversion avoidance, inter-thread synchronization, timeouting and aborting threads, and managing interrupts (see Section 2).

In .NET, thread priorities are not fixed. The MSDN .NET class library in its `ThreadPriority` Enumeration section states: “The operating system can also adjust the thread priority dynamically as the user interface's focus is moved between the foreground and the background” [.NETLIB]. Moreover, according to [Lutz03], priority inheritance does not seem to exist in the .NET lock mechanism. As for the ability to abort a running thread, class `System.Threading.Thread` contains an `Abort()` method, which throws a `ThreadAbortException` [Richter02]. However, the MSDN .NET class library states, “calling this method *usually* terminates the thread”. This is because throwing the above exception causes all `finally` clauses of currently entered `try` blocks to execute. These might perform unbounded computations, causing a thread to execute indefinitely. Any real-time abort mechanism introduced should guarantee a bounded latency.

Writing interrupt handlers could be done through unmanaged code executing outside the CLR, but this is besides our interests. A mechanism has to be provided in the Framework itself.

Following we specify additional facilities that are also considered to be vital for real-time performance:

- A real-time clock, with a specified granularity (e.g. compatible with POSIX).
- Thread abstractions, like periodic, aperiodic, and sporadic threads. Thread abstractions should follow the object-oriented, dynamic, and nested threading model, that .NET supports.
- Although thread priorities exist in .NET, the value set is completely inadequate for real-time systems (5 priorities). An extended set of priorities (e.g. 256) has to be provided.

Furthermore, RT.NET should try to address specialized real-time programming needs by tackling key technical issues, like: small memory footprint, fast interrupt response latency, breadth of general purpose operating system services, and host-target cross development tools. Differentiations based on technical issues as these cover a wide range of real-time systems and would make RT.NET more versatile in this marketplace. Especially for memory footprint [Lutz03] shows that C# is up to 50% more consuming than C, making its use for embedded systems problematic.

Finally, as mentioned, RT.NET should support advancement of the state of the art. Today much focus is placed on the following issues:

- Provision for on-line feasibility (schedulability) analysis for the acceptance of new real-time activities, and execution-time analysis.
- Independent resource-need determination of real-time components and negotiation of resources with the system.
    - Processor bandwidth-latency guarantees (processor reserves).
    - Disk I/O bandwidth reservation.

○ Network rate-latency guarantee.	[Liu03]

4.2 Profiles and the Core

The concept of a real-time Core and additional profiles could easily be applied to .NET, since there already exists the idea of profiles. The CLI standard defines profiles to be sets of "libraries grouped together to form a consistent whole that provides a fixed level of functionality" [ECMA02]. There are two profiles defined in the standard, the Kernel Profile and the Compact Profile (Figure 4 [ECMA02]). The Kernel Profile is "the minimal possible conforming CLI implementation" consisting of the Base Class Library and the Runtime Infrastructure Library, while the Compact Profile is "designed to allow implementation on devices with only modest amounts of physical memory" yet providing more functionality than the Kernel Profile alone.

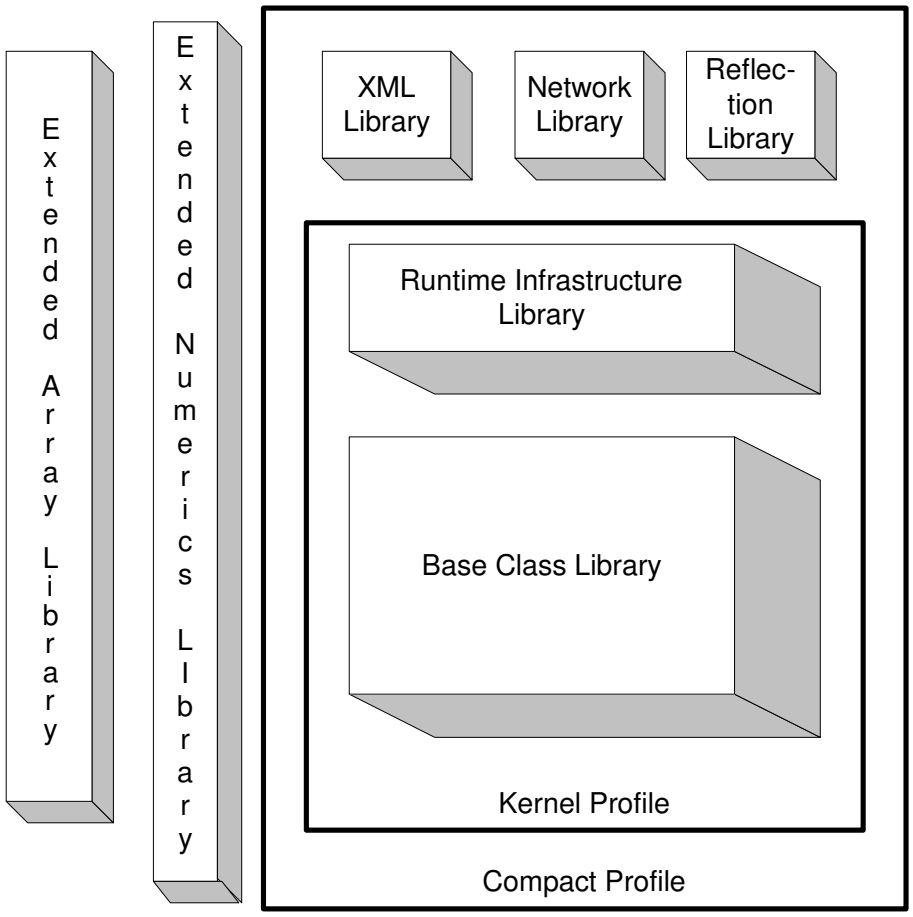

Figure 4: The CLI Kernel and Compact Profiles

Having that in mind, an implementation could follow either of the two deployments. However, the second deployment in Figure 3, a new Real-Time CLR, combining all the traditional CLR functionality with the real-time Core functionality, would be more desirable. Building a real-time-augmented CLR should be more convenient, since we can use the existing CLR code and modify it to add real-time functionality. CLR code is freely available through the Shared Source Common Language Infrastructure initiative (SSCLI, aka ROTOR) [ROTOR]. In addition, following the second deployment makes it easier to specify the relationship and interoperation between traditional CLR threads and real-time CLR threads. For example, there is no need for a separate API for communication between the two kinds of threads, as is the case with RTCE. Irrespectively of the method of deployment, the functionality offered should be of a high-level similar to RTSJ. This is both for increased programmer productivity and to counterweight RTSJ. Profiles can be implemented on top of the core real-time CLR functionality. More specifically, each additional profile would include the Kernel Profile and also one or more additional libraries.

4.3 Core Requirements

The most important contribution of the NIST report is the requirements for the specification of the real-time Core functionality. These requirements constitute the basis for any implementation. This section translates all nine requirements for RT.NET, using most of them verbatim and keeping the same numbering.

- **Core Requirement 1**: The specification must include a framework for the lookup and discovery of available profiles.
- **Core Requirement 2**: Any garbage collection that is provided shall have a bounded pre-emption latency. The pre-emption latency is the time required to pre-empt garbage collection activities when a higher priority thread becomes ready to run. The specification must clearly define the restrictions that are imposed by pre-emption.

An important derived requirement here is that for every language or library feature in .NET, one must be able to quantify the upper limit on the memory resources used. Also, if a GC is used, the specification must state which of the .NET functionality is usable by real-time threads so that their execution does not interfere with the state of the heap. This will ensure that the garbage collector will not be mistakenly called. Finally, the GC overhead on the application, if any, must be quantified. In general, we can say that garbage collection will either be real-time or will be disabled altogether (see Section 3.5, Goal 8).

- **Core Requirement 3**: The RT.NET specification must define the relationships among real-time .NET threads at the same level of detail as is currently available in existing standards documents. An example of the minimal requirements for specification would be POSIX 1003.1b. Examples of these relationships include thread scheduling, wait queue ordering, and priority inversion avoidance policies.
- **Core Requirement 4**: The RT.NET specification must include APIs to allow communication and synchronization between managed real-time .NET threads and non-managed threads.
- **Core Requirement 5**: The RT.NET specification must include handling of both internal and external asynchronous events. The model must support a mechanism for executing .NET code in response to such events. The mechanism should fit in well with existing mechanisms (such as `wait/pulse`). ("External" is taken to mean interrupts or OS signals)
- **Core Requirement 6**: The RT.NET specification must include some form of asynchronous thread termination. This asynchronous thread termination must have the following properties:
  1. By default a thread cannot be aborted.
  2. The target code determines where it can be aborted.
  3. When a thread is aborted:
     a. all locks are released, and
     b. `finally` clauses execute on the way out when the thread is being aborted.

4. No facility for aborting uncooperative code need be provided. The termination shall be deferred if the code currently executing has not indicated a willingness to be terminated.
5. Mechanisms must be provided that allow the programmer to insure data integrity.

- **Core Requirement 8**: The RT.NET specification must provide a mechanism to allow code to query whether it is running under a real-time .NET thread or a non-real-time .NET thread.
- **Core Requirement 9**: The RT.NET specification must define the relationships that exist between real-time .NET, non-real-time .NET threads and non-managed threads. It must provide mechanisms for communication and sharing of information between real-time .NET threads and non-real-time .NET threads.

To complement this requirement it is important to specify that traditional .NET threads must run as non-real-time threads. Also, sharing and communications protocols must have known tight upper bounds or some other form of predictability on blocking delays (see [NIST99] for a definition of "predictability"). Finally, the specification must describe the relationship between real-time threads and all the other thread types (non-real-time .NET, non-.NET real-time, non-.NET non-real-time), including priorities and scheduling, synchronization, sharing resources, other processes, budgets (memory, CPU, etc), and protections and privileges. Additionally, we could argue that for the purposes of scheduling (execution eligibility) RT.NET could treat non-real-time .NET threads as real-time .NET threads with the lowest eligibility.

- **Core Requirement 7**: The RT.NET core must provide mechanisms for enforcing mutual exclusion without blocking. This requirement does not imply that a real-time thread should be allowed to disable/enable interrupts to achieve these results. The specification must require that the mechanisms do not allow a non-real-time thread to gain complete control of the machine. Specifically, the scheduler will continue to dispatch threads and interrupt handlers, other than the one possibly attached to the thread using the non-

blocking mutex, which will continue to execute. The specification should take care to minimize risks to the system integrity.

As [NIST99] states, the initial thought for mutual exclusion without blocking was the construction of atomic sequences of instructions. Masking interrupts, although an obvious mechanism, is avoided, since the dispatcher is required to continue switching threads and servicing interrupts. So is a "conditional acquire lock" (like the `sem_trywait()` function in POSIX). One suggestion is a global lock shared by all threads. It has to be said that the NIST report is vague on this point. Perhaps the idea here was to have something like lock-free synchronization mechanisms, as in a thread that executes a certain piece of code, checking to see whether some other higher priority thread got to execute the same code before the lower priority thread finished. In the case that a higher priority thread did run the lower priority one executes the code again [Anderson97].

4.4 Additional Core Requirements

The following are *additional* core requirements for RT.NET:

- **Core Requirement 10**: Any real-time functionality added to .NET shall be accessible to *any* language targeting the .NET Framework (i.e. it shall conform to the Common Language Specification (CLS)).
- **Core Requirement 11**: RT.NET shall provide mechanisms for asynchronous transfer of control (ATC).

Core Requirement 10 is .NET-specific and has to be included to cater for all those languages that might want to make use of RT.NET features. Although we will only be considering C# for our implementation there might be a future interest in using another language, e.g. Ada, to leverage the real-time facilities of RT.NET. Core Requirement 11 stems from the nature of real-time systems, which usually involve interaction with the real world. As a result of that RT.NET might have to enforce a sudden change in the flow of execution, which calls for the need to have a high-level

ATC mechanism. This mechanism is considered to be different from that required to abort a thread.

4.5 Goals and Derived Requirements

The final section of the NIST report presents a list of thirteen goals and associated derived requirements. These are mainly intended for future use. Ten of these goals have been translated and are used for RT.NET, with minimal changes in their wording. For each RT.NET goal the corresponding goal of the NIST report is given within parentheses.

- **Goal 1**: RT.NET should allow any desired degree of real-time resource management for the purpose of the system operating in real-time to any desired degree (e.g. hard real-time, and soft real-time with any constraints, collective timeliness optimisation criteria, and optimality/predictability tradeoffs). (NIST Goal 1)
- **Goal 2**: Subject to resource availability and performance characteristics, it should be possible to write RT.NET programs and components that are fully portable regardless of the underlying platform. (NIST Goal 3)
- **Goal 3**: RT.NET should support workloads comprised of the combination of real-time tasks and non-real-time tasks. (NIST Goal 4)
- **Goal 4**: RT.NET should allow real-time application developers to separate concerns between negotiating components. (NIST Goal 5)
- **Goal 5**: RT.NET should allow real-time application developers to automate resource requirements analysis either at runtime or off-line. (NIST Goal 6)
- **Goal 6**: RT.NET should allow real-time application developers to write real-time constraints into their software. (NIST Goal 7)
- **Goal 7**: RT.NET should allow resource reservations and should enforce resource budgets. The following resources should be budgeted: CPU time, memory, and memory allocation rate. (NIST Goal 8)

NIST Goal 8 was accepted with serious reservations. It could be substituted with one of its derived requirements: RT.NET infrastructure should allow negotiating

components to take responsibility for assessing and managing risks associated with resource budgeting and contention. This means that each component of an RT.NET application should be able to specify its resource needs in such a way so as to get the best treatment from the runtime environment, taking into consideration any information passed to it from the environment, regarding system contention.

- **Goal 8**: RT.NET must provide real-time garbage collection when garbage collection is necessary. GC implementation information must be visible to the RT.NET application. (NIST Goal 10)
- **Goal 9**: RT.NET should support straightforward and reliable integration of independently developed software components (including changing hardware). (NIST Goal 11)
- **Goal 10**: RT.NET should be implementable on operating systems that support real-time behaviour. (NIST Goal 13)

4.5.1 Rejected Goals

The following goals present difficulties in adapting them for RT.NET:

- **NIST Goal 2**: Support for RTJ specification should be possible on any implementation of the complete Java programming language.

NIST Goal 2 seems vague, since it does not make absolutely clear whether it talks about the *Java Language Specification* or the *Java Virtual Machine Specification*. Its derived requirements talk about scaling to large or small memory systems, making it more likely to be talking about the virtual machine. However, we cannot safely interpret this goal for .NET.

- **NIST Goal 9**: RTJ should support the use of components as “black boxes”; including such use on the same thread.

NIST Goal 9 did not achieve consensus, so we are not including it for RT.NET.

- **NIST Goal 12**: RTJ should be specified in sufficient detail to support (and with particular consideration for) targeting other languages, such as Ada.

Finally, NIST Goal 12 refers to Java targeting other languages (e.g. Ada). This is an intrinsic feature of .NET, and there actually exists an Ada port for .NET under the name A# (a-sharp) [A#]. Consequently, there is no reason for this goal in .NET.

5 Conclusions

In this work we examined the NIST Real-Time Requirements for Java report [NIST99] under the prism of a Real-Time .NET Framework notion. The NIST report, although focusing on Java technology, does provide a sound starting ground for the development of real-time requirements for .NET. Based on these, a real-time specification for .NET is feasible and is part of our future goals.

6 References

[.NETLIB] *.NET Framework Class Library*, MSDN, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/cpref_start.asp?frame=true

[A#] Carlisle, Martin C., Sward, Ricky E., and Humphries, Jeffrey W., "*Weaving Ada95 into the .NET Environment*", Dept. of Computer Science, US Air Force Academy, 2002, found at http://www.usafa.af.mil/dfcs/bios/mcc_html/a_sharp.html

[Anderson97] Anderson, James H., Ramamurthy, Srikanth, and Jeffay, Kevin, "*Real-Time Computing with Lock-Free Shared Objects*", University of North Carolina, 1997.

[Cornelius02] Cornelius, Barry, "*Comparing .NET with Java*", IT Service, University of Durham, 2002.

[ECMA02] Standard ECMA-335, "*Common Language Infrastructure*", 2nd Edition, ECMA, December 2002.

[Kwon03] Kwon, J., Wellings, A.J., and King, S., "*Ravenscar-Java: A High-Integrity Profile for Real-Time Java*", ACM Java Grande – ISCOPE 2002

[Liu03] Liu, Jane W. S., "*Predictability of Real-Time Software on Commodity Platforms*", PowerPoint presentation, ESSES 2003.

[Lutz03] Lutz, Michael H., and Laplante, Phillip A., "*C# and the .NET Framework: Ready for Real Time?*", IEEE Software Magazine, January/February 2003 (vol. 20, No. 1), p. 74-80.

[MSDN] MSDN Library, .NET Framework Developer's Guide, "*Overview of the .NET Framework*".

[NIST99] National Institute of Standards and Technology, "*Requirements for Real-time Extensions For the Java Platform*", September 1999, taken from http://www.nist.gov/rt-java

[Puschner01] Puschner, P., and Wellings, A.J, "*A Profile for High-Integrity Real-Time Java Programs*", IEEE ISORC, 2001

[Richter02] Richter, Jeffrey, "*Applied Microsoft .NET Framework Programming*", Microsoft Press, 2002.

[ROTOR] Shared Source CLI home page at http://www.sscli.net/

[RTCE00] "*Real-Time Core Extensions*", J Consortium, September 2000, available at www.j-consortium.org

[RTSJ00] The Real-Time for Java Expert Group (Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M., and Belliardi, R.), "*The Real-Time Specification for Java*", Addison-Wesley, June 2000, available at www.rtj.org

[Struys03] Struys, Maarten, and Verhagen, Michel, "Real-Time Behaviour of the .NET Compact Framework", MSDN, 2003.