Toward a Unified Standard for Worst-Case Execution Time Annotations in Real-Time Java

Trevor Harmon and Raymond Klefstad

University of California, Irvine
Dept. of Electrical Engineering and Computer Science
Irvine, California 92697-2625 USA
{tharmon, klefstad}@uci.edu

Abstract

As real-time systems become more prevalent, there is a need to guarantee that these increasingly complex systems perform as designed. One technique involves a static analysis to place an upper bound on worst-case execution time (WCET). This temporal analysis cannot be made automatic and normally requires source annotations to assist a WCET analysis tool.

At the same time, there is a growing interest in using Java for real-time systems. Several WCET analysis prototypes for Java have been created, and more are under development. Each relies on a competing and incompatible convention for annotations, resulting in portability problems and duplication of effort.

We propose that Java's own annotation mechanism should be used to address such issues. These built-in annotations provide a common platform for WCET analysis, improving portability and reducing the effort necessary to create these vital tools. We examine the features that make Java's annotation standard attractive for WCET analysis, then discuss its current failings and make recommendations for future improvements.

1. Introduction

Knowing the *worst-case execution time*, or WCET, of tasks can provide great confidence and predictability in real-time systems. Unfortunately, WCET analysis is often neglected or merely "guesstimated." Although WCET research is improving, and commercial tools have helped, WCET analysis remains limited and *ad hoc*. Even when an analysis is performed, it may not provide a guarantee (leading to poor confidence), or the results may be overly pessimistic (leading to underutilized resources).

1-4244-0910-1/07/$20.00 ©2007 IEEE.

To understand why the worst-case execution time is so important, consider the major real-time scheduling algorithms. All of these algorithms *require* knowledge of the WCET. To be specific:

Rate-monotonic This algorithm assumes that no task has a WCET longer than its deadline. Furthermore, its feasibility analysis requires WCET as input: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(\sqrt[n]{2} - 1\right)$, where U is processor utilization, n is the number of processes, T_i is the release period, and C_i is the WCET.

Earliest Deadline First (EDF) This scheduler does not require WCET knowledge at run-time. However, EDF is unstable (one late job causes many other jobs to be late), so a guarantee against overloading is vital. The only way to make this guarantee is to perform an acceptance test: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$, where the variables have the same meanings as above. Again, note that WCET is part of the equation.

Least Slack Time (LST) The slack (also called "laxity") of a real-time task is defined as $d - t - c$, where d is the deadline, t is the current time, and c is time required to complete the remaining portion of the job. Determining c requires knowledge of the WCET.

Time-triggered Also known as *clock-driven* or *time-driven*, this approach depends on a static schedule computed prior to execution. The static schedule dictates an exact moment in time for each task's release. In such a scheme, task n cannot be scheduled until task $n-1$ completes; otherwise, two tasks could be released at the same time. Preventing this failure requires knowledge of each task's WCET.

Note that *all* of the above algorithms depend on WCET. Thus, even if the underlying operating system implements a solid, carefully crafted real-time schedul-

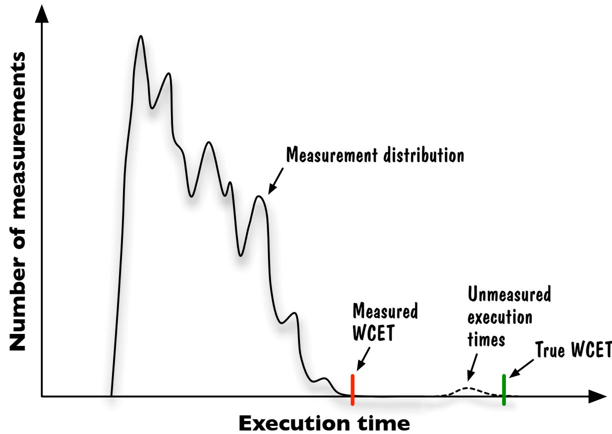

Figure 1. This histogram of execution time for a fictional real-time task illustrates the weakness of measurement when dealing with worst-case execution time.

ing algorithm, making any guarantee about the predictable behavior of a task is impossible unless its WCET is known. And, as Figure 1 shows, an upper bound on the WCET cannot be obtained through measurement; a static analysis must be performed.

2. The Challenge of WCET Analysis

The essential question, then, is why more influential research has not been produced despite two decades of work in WCET.[1] One explanation comes from Kirner and Puschner [9], who argue that industrial-strength WCET tools are simply too difficult to implement.

2.1. Implementation Difficulties

One of the factors behind this implementation difficulty is the modern CPU. Architectural advancements in RISC processor design, such as very long pipelines and complex multi-level caches, have focused on making the average case as fast as possible. Unfortunately, the shrinking of this average has not come without cost: While the average may be small, its standard deviation has grown large, resulting in large (and overly pessimistic) worst-case execution times.

Hardware is not the only difficulty, of course. The software platform is another reason for the lack of good tools for WCET analysis. Today, most real-time systems are developed in C and, to a lesser extent, Ada. Because the compilers for these languages vary, there is no common intermediate representation (IR) for WCET analysis tools to target. GCC's Register Transfer Language, for example, is incompatible with the Intel compiler's IR. Furthermore, these IRs may change across compiler versions, they may suffer from limited documentation (if any), and they are subject to change with each new compiler version.

The changing nature of IR magnifies the difficulty of building WCET analysis tools. Because there is no clean, consistent separation between high-level source code and low-level machine code, tools must be able to perform a complete top-to-bottom analysis. This includes parsing source code, constructing a control flow graph, mapping the basic blocks to machine code, analyzing each basic block according to a model of the target processor, and so on.

To make this process tractable, most tools offer little flexibility, usually locking themselves to a particular hardware architecture. It is also common to ignore high-level source code altogether, operating at the machine code level. Mapping this machine code analysis back to the high-level source code, which is necessary to act on the information provided by the tool, is typically cumbersome and unintuitive. The aiT tool,[2] for instance, displays its analysis in assembly language, leaving the programmer to mentally map the jumble of mnemonics and hexademical numbers back to the source code language of choice.

This situation has led developers, as well as researchers in the WCET field, to seek a better platform on which to build real-time systems and tools.

2.2. Java as a Catalyst

As a platform with no temporal guarantees and underspecified threading semantics, Java seems an unlikely choice for real-time systems development. Yet there are a number of reasons why an increasing number of researchers are looking to Java for real-time environments:

Bytecode Java's bytecode provides an inherent modularization of static analysis tasks, as illustrated in Figure 2. For example, high-level WCET tools for Java can ignore any timing aspects below the bytecode level. Separate low-level tools, perhaps written by other vendors, can then complete the analysis once the target architecture is known. This sort of modularization helps solve the software complexity problem raised in Section 2.1. Bytecode also acts as a common, well-specified intermediate representation that does not vary with different compiler versions and vendors (unlike the situation with C and Ada).

[1] Kligerman's and Stoyenko's 1986 paper [10] is generally considered the first publication to address the problem of WCET.

[2] `http://www.absint.com/ait/`

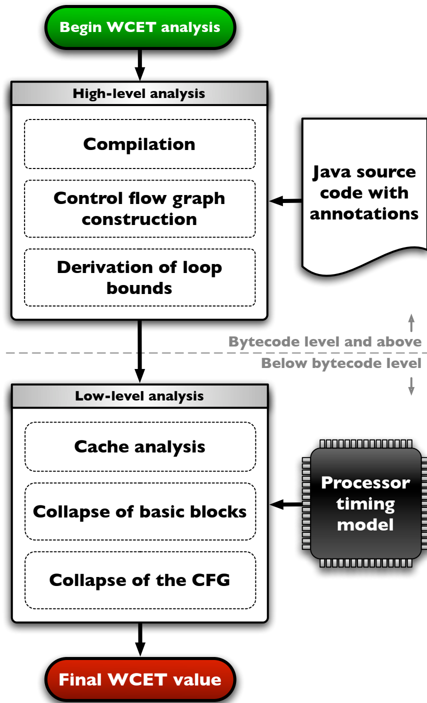

Figure 2. This sketch of the WCET analysis process shows the clean separation between high- and low-level analysis that Java bytecode provides.

```
1  public static void main(String[] args) {
2    int max = Integer.parseInt(args[0]);
3    ...
4  }
5
6  private static void sort(double[] a) {
7    // @WCET loopMax=9
8    for (int i = 0; i < max-1; i++) {
9      // @WCET loopMax=9
10     for (int j = 0; j < max-1-i; j++) {
11       if (a[j+1] < a[j]) {
12         double tmp = a[j];
13         a[j] = a[j+1];
14         a[j+1] = tmp;
15       }
16     }
17   }
18 }
```

Figure 3. WCET tools require knowledge about the constraints under which a program will run. Here, the developer "knows" the input size will never be larger than 10 and has inserted WCET annotations accordingly.

Java processors Java processors, whose native instruction set is Java bytecode, help solve the hardware complexity problem raised in Section 2.1. These chips eliminate the operating system and virtual machine, making WCET analysis far simpler. In addition, research has produced a Java processor, called JOP, with a short four-stage pipeline and a predictable instruction cache designed for real-time systems [15]. Such chips are easy targets for tight WCET analysis.

Productivity Real-time systems are becoming increasingly complex and widespread. To keep up with the growing demand for these sophisticated systems, developers need to become more productive, and Java is recognized as a more productive language than C [11].

3. WCET Annotations

While Java provides simplicity, WCET analysis is still far from trivial. The root of the problem has been known since 1936 when Alan Turing proved that, given an arbitrary program, a decision as to whether the program finishes or will run forever does not exist [16]. The consequence for WCET analysis is that no tool can examine an arbitrary real-time Java program and derive its WCET bounds. As shown in Figure 3, the WCET may depend on knowledge that only exists at run-time, stifling any attempt at static analysis.

WCET bounds may be undecidable in general, but real-time systems are hardly the arbitrary programs described by Turing. In a real-time environment, the developer is normally able to (and, in many cases, must) exercise careful control over input parameters and program complexity. As a result, certain assumptions can be provided to a WCET analysis tool that make its job tractable. These assumptions usually take the form of *annotations*.

For example, turning again to Figure 3, we see that the developer knows something about the run-time environment and is able to make an assumption: The maximum array size (that is, the `args[0]` parameter) for any execution of this program will never be larger than 10. The developer was thus able to *annotate* the source code on lines 7 and 9 with the maximum possible iteration of each loop. By propagating this special knowledge from developer to analysis tool, annotations provide much tighter WCET bounds than would otherwise be possible.

3.1. Prior Work in WCET Annotations for Java

Given that WCET annotations have been used for many years in real-time C and Ada programs, one might expect a standard, or at least a *de facto* convention, to materialize. In reality, a number of competing and incompatible styles of annotation have been devel-

oped for real-time Java. This section provides a brief survey of those styles. It includes only the research literature and does not count commercial tools and other annotation styles that may be found "in the wild."

The earliest work in WCET annotations for Java comes from Bernat, who proposed a portable WCET analysis tool [1]. In this case, *portable* refers to language portability: The tool was designed for analyzing Java, C, Ada, and any other language that could be translated to Java bytecode. To achieve such portability, source code must invoke methods in a predefined class whenever a WCET annotation is required. (See Table 1 for an example.) Compared to traditional annotations, this style mingles non-functional metadata (that is, the WCET information) with the normal source code statements, making programs more difficult to read.

The XRTJ project [8] implemented WCET annotations in a more traditional way. All annotations appear as comments with the characters `//@` for single lines and `/*@ ... @*/` for multiple lines [6]. The XRTJ compiler parses these lines and writes them to an XAC (Extensible Annotation Class) file, an XML-like text file that is paired with its class in a real-time Java program. The XRTJ analyzer then reads each XAC file to determine loop bounds, timing modes, and other details necessary to derive the WCET. Hu later extended the XAC format to capture dynamic dispatch (i.e., polymorphism) semantics [5].

In more recent work, Schoeberl and Pedersen implemented a WCET analyzer for JOP [14]. This tool introduced yet another syntax for annotations. Although it still used the same `//@` start marker, the syntax of the annotations differed from the XRTJ project, making the tools incompatible with each other.

Finally, it should be noted that neither of the standards for real-time Java, RTSJ [2] nor RTCE [3],[3] provides any mechanism whatsoever for annotations, despite the importance of WCET as outlined in Section 1.

3.2. A Lack of Standards

Clearly, no single convention for WCET annotations has taken hold, resulting in a contradiction of Java's mantra of "write once, run anywhere." Real-time Java programs designed for one WCET analysis tool must be rewritten for other tools. The lack of a single annotation standard also makes the tools themselves more difficult to implement. Even if they all understood `//@` as a starting token for an annotation, a number of unresolved issues remain:

[3]RTCE is now defunct, supplanted by the RTSJ.

- What is the syntax for the string following `//@`?
- After an annotation is parsed, where is the information stored and how do lower-level tools retrieve it?
- Is there a formal specification to ensure compatibility? Who ratifies it and who maintains it?

4. A Standard for Java Annotations

In the non-real-time Java domain, a similar situation had already taken place. The Java Modeling Language (JML), the ESC/Java[4] static checker, XDoclet, and various other software offered custom, incompatible annotation systems for Java source code. In addition, Java's frameworks for server software development, such as Enterprise JavaBeans, had become increasingly complex due to an explosion of metadata.

To solve both problems, the Java community proposed a standard framework for annotations. The basic idea was to encapsulate common code patterns into single statements—annotations—embedded directly into the source code. Rather than manage source code and metadata separately, taking pains to keep them in sync, annotations could be attached directly to the programming constructs they describe. This is a much more powerful and convenient approach. For example, annotations allow the canonical getter/setter methods:

```
private int sensorReading;

public int getSensorReading() {
  return sensorReading;
}

public void setSensorReading(int sensorReading) {
  this.sensorReading = sensorReading;
}
```

to be condensed to:

```
@property int sensorReading;
```

Note that, unlike the approaches described in Section 3, these annotations are not comments. Instead, they are first-class objects in the Java language, requiring special compiler support. They can have parameters and must conform to type-checking rules, for example. Such a substantial change to the language demanded formal review under the Java Community Process (JCP): a public, cooperative system for adopting new technologies as official Java specifications.

The proposed annotation framework was submitted to the JCP as JSR-175[5] and met final approval in September 2004. That same month, Sun released Java

[4]`http://en.wikipedia.org/wiki/ESC/Java`

[5]JCP proposals begin life as a numbered Java Specification Request, or JSR. All JSRs mentioned in this paper can be downloaded from `http://jcp.org/`.

Table 1. A sample of WCET annotation styles in real-time Java

Tool	Examples	Comments

WCETAn [1]	`WCETAn.Scope S = new WCETAn.Scope();`
	`WCETAn.Loopcount(20);`
	`WCETAn.Define_Path(1);`
		Implemented as method calls instead of comments for source language portability.

Skånerost [13]	`/*$ path-bound 10 */`
	`/*$ time-bound 25ms */`
	`/*$ loop-bound 100 */`
		Departs from the near-universal convention of using `@` as the start token in annotations, opting for `$` instead.

XAC [6]	`//@ Loopcount(100)`
	`//@ Mode(Quick_Mode)`
	`//@ UseWCET(AirTempSen.AccessSensor(V))`
		Traditional comment-based annotations. Has been extended to handle polymorphic method calls.

WCA [14]	`//@WCA loop=10`	Currently supports loop bound constants only.

5, Standard Edition, which included support for JSR-175 annotations. A new edition of the Java Language Specification [4] was published the following year to formalize these annotations and ensure that tools and libraries using them would remain compatible with each other. Today, annotations are a standard, well-defined part of the Java platform.

4.1. Storing Java Annotations

One of the advantages of Java annotations,[6] and a key difference versus existing WCET annotation frameworks, is that the annotation data is stored directly in class files.[7] Embedding annotations within classes simplifies code management because a separate file format for metadata need not be maintained. Also, existing mechanisms for storing, distributing, and deploying class files can readily be used for annotation data. For instance, a build script that packages a class library into a JAR file (Java ARchive) will automatically package annotations, as well.

The annotation standard achieves this simplicity by building upon an existing Java mechanism for bundling metadata with class files. Known as *class file attributes*, this mechanism has been part of the Java virtual machine specification since its inception. It defines an area at the end of the class file for storing any kind of structured data, such as the line number table, the source file name, and even the Java bytecode itself.[8]

Java compilers and other code generators are permitted to emit class files containing new attributes, and Java virtual machine implementations are prohibited from refusing to load class files simply because of the presence of some new attribute. Thus, class file attributes are *extensible* and may support new attributes at any time without sacrificing backward compatibility.

The JSR-175 specification takes advantage of this extensibility by using it to store annotations. It reserves their names to avoid conflicting with other attributes, and it formalizes their structure so that third-party tools know how to access them.

4.2. Creating Java Annotations

Of course, programmers need not write to these attributes directly. The Java compiler generates them automatically when compiling Java source code containing annotations. For example, Java 5 includes a few built-in annotation types such as `SuppressWarnings`:

```
@SuppressWarnings({"deprecation"})
public void myMethod() { ... }
```

Compiling this code results in a class file with a `RuntimeInvisibleAnnotations` attribute pointing to the `SuppressWarnings` annotation type. As expected, compilers will not print deprecation warnings for the annotated method. Furthermore, the "invisible" specifier tells virtual machines not to load the attribute into memory because it is useful only at compile time.

Custom annotations can be created easily, as well. Imagine a real-time control system that must respond to a sensor change and actuate a motor within 50 milliseconds. This requirement could be encoded as an annotation, perhaps called `MaxAllowableWCET`, and attached directly to the very method that handles the response. WCET tools analyzing the method could then print a warning if the calculated WCET exceeds the specified WCET.

Creating this sort of annotation requires defining an *annotation type* that resembles a traditional Java in-

[6]From this point forward, "Java annotations" refers to the JSR-175 standard.

[7]We are not the first to propose embedding WCET annotations in class files. In [7], Hu briefly remarked that XAC annotations [6] could be stored there.

[8]For a complete description of class file attributes, refer to Section 4.7 of the Java Virtual Machine Specification.

```
for (Method m :
   Class.forName(args[0]).getMethods())
{
  if (m.getName().startsWith(args[0])) {
    if (!m.isAnnotationPresent(
          MaxAllowableWCET.class)) {
      System.out.println(
        "Error: A WCET annotation is missing");
    }
  }
}
```

Figure 4. This example code uses Java's Reflection API to verify that all methods are annotated with the `MaxAllowableWCET` type described in Section 4.2.

terface:

```
@Target(ElementType.METHOD)
public @interface MaxAllowableWCET {
  double seconds();
}
```

Here, the `seconds` method declares a parameter for the annotation type, while the `Target` annotation (actually a meta-annotation, since it annotates an annotation) tells the compiler that this annotation type should be used only on method declarations.

After compiling the annotation type, using it is a simple matter:

```
@MaxAllowableWCET(seconds=0.05)
public void actuateMotor() { ... }
```

4.3. Reading Java Annotations

Tools such as WCET analyzers need to read the annotations programmers have written, and Java provides a facility for this. Java 5 includes a new package, `java.lang.annotation`, and an expanded Reflection API to access annotations. Figure 4 shows a simple example of how `MaxAllowableWCET` annotations can be detected using this API. (For the example to work, the `MaxAllowableWCET` annotation type must have a retention policy of `RUNTIME` so that it is visible to the virtual machine. See Table 2 for a list of retention policies.)

In addition to this built-in mechanism, a number of third-party frameworks have been developed that can assist in reading and manipulating Java annotations: Annogen, ASM,[9], and Javassist, to name a few. All of these work with the same Java annotation standard, so unlike the current situation with WCET analysis, annotations created by one tool can be accessed by another.

[9] http://asm.objectweb.org/

Table 2. Java annotation retention policies

Policy	Description
SOURCE	The annotation is not stored in the class file; it is simply discarded at compile-time.
CLASS	The annotation is stored in the class file but is not loaded into memory.
RUNTIME	The annotation is stored in the class file and loaded into memory at run-time so that it is visible to Java's Reflection API.

4.4. Weaknesses of Java Annotations

While standardization, portability, and validation are the greatest strengths in Java's annotation facility, it also has some serious drawbacks. Chief among them is a restriction on where annotations can be placed in the source code: They can act only as declaration modifiers. They cannot annotate method calls, loops, and other program elements. For example, the following code is illegal:

```
@LoopBound(max=10)
while (!bufferEmpty) { ... }
```

Another limitation is that Java's Reflection API can only read annotations that have been stored in class files. Source code with annotations must first be compiled before the annotations can be read. A more serious implication is that if an annotation type has a `SOURCE` retention policy, it cannot be accessed at all without third-party tools that are able to parse Java source code.

The Java community has recognized these weaknesses and responded with a series of proposals submitted to the JCP. For example, JSR-269 provides a standard API for interacting with annotation processors such as Annogen. It also provides an interface for processing annotations in source files. The proposal was approved and will be implemented in Java 6.

Another proposal, JSR-308, allows annotations to be placed where types are used, not just where they are declared. For instance:

```
myString = (@NonNull String) myObject;
```

The proposal has been approved and will likely be implemented in Java 7.

5. Applying Java's Annotation Standard to WCET

Compared to existing comment-based annotation systems in WCET tools, which are proprietary and

sometimes implemented *ad hoc*, Java's built-in annotations offer an attractive alternative. In particular:

Standards-based Java's annotation mechanism solves the standards issues raised in Section 3.2.

Validation Unlike most comment-based systems, Java annotations are highly structured and type-safe. For instance, the Java compiler can guarantee that an annotation designed for methods is not accidentally used to annotate other program constructs.

Convenience With WCET annotations implemented as Java annotations, timing information and code can be bundled in the same class file. This simplifies code management and distribution. For example, a class library for real-time systems can include timing information in the library itself. No extra files and no new file formats are necessary.

Tool support Because Java's annotation mechanism is relatively mature and ships with every copy of the Java run-time, it enjoys broad support from a variety of APIs that make working with annotations easier. WCET analysis tools are therefore easier to implement since they can be built upon these existing APIs.

Comment-based WCET annotations described in the literature do not have such features. For this reason, we propose that WCET analysis tools should adopt Java's annotation standard. Doing so would likely shorten the development cycle of WCET analysis tools, leading to more prototypes and an overall improvement in real-time research.

Unfortunately, Java's standard is not yet a drop-in replacement for existing annotation frameworks. One of the issues is a concern for embedded systems. (Embedded systems are not directly related to WCET analysis, but real-time systems are often embedded systems, too, so the domains overlap.) These systems normally have stringent resource requirements, so the Java class files deployed on them must be as small as possible. However, Java annotations are stored in these class files, taking up valuable memory even though the annotation information is only required at design time.

One way to solve this problem is to change the retention policy of the annotation to `SOURCE`. This will prevent the annotation from being stored in the class file, but it will also destroy the benefit of bundling annotation data with the code itself. A better workaround is to leave the annotation at the default `CLASS` retention level but use a tool to strip the annotations from the class files before deploying them to the embedded device.

A more critical failing of Java's annotation standard cannot be mitigated so easily. Namely, the restriction on where annotations can be placed is a major impediment toward using standard Java annotations in WCET tools. As a result, none of the existing WCET annotation styles listed in Table 1 can be fully translated to Java annotations.

```
@LoopBound(max=10) int i;
for (i = 0; i < 10; i++) {
  if (b) {
    @LoopBound(max=4) int j;
    for (j = 0; j < 4; j++)
      val *= val;
  }
  else {
    @LoopBound(max=7) int k;
    for (k = 0; k < 7; k++)
      val += val;
  }
}
```

Figure 5. This altered program segment from JOP's test suite shows how WCET annotations might be implemented according to the Java standard.

In certain scenarios, however, an approximation is possible. For example, Figure 5 shows a test method from the JOP project [14] that we have re-written using the following Java annotation type:

```
public @interface LoopBound {
  int max();
}
```

Note that instead of defining loop bounds next to the loop constructs, Figure 5 defines them at the loop variable declarations. Although we have verified that JOP's WCA tool can run successfully with these modifications, restricting loop bounds to variable declarations is too limiting. It is also unintuitive and awkward.

We note that other researchers have encountered the same obstacles in Java annotations while working independently on other problems. In fact, JSR-308 was born as a result of these obstacles [12]. It loosened the restrictions on where annotations may be placed. Unfortunately, the proposal did not relax them far enough for the purpose of WCET annotations, which would require at a minimum placement of annotations directly on loop constructs (`if`, `while`, and `do/while`).

6. Future Work

Although Java annotations are not currently suitable as WCET annotations, the standard is not far from becoming a virtually ideal replacement. Two changes are necessary for this to happen.

First, the standard should be modified so that annotations can be placed on loops and basic blocks. JSR-308 is already moving the standard in this direction. Indeed, the original proposal [12] commented that such a change would be useful for concurrency and atomicity. (It did not mention WCET analysis.) In addition,

members of the Expert Group who voted for the proposal agreed that relaxing the standard in this manner would be beneficial. For instance, Intel Corporation adding the following comment to its vote:

> *This note confirms our understanding that the scope of the JSR includes providing for annotations on loops and blocks if the Expert Group decides to include that after evaluation. The JSR itself should be updated to make it clear this is in the scope.*

Second, type definitions and a naming convention for WCET annotations must be established. Java annotations would do little good if one tool recognized `@LoopBound` while another tool expected, say, `@LoopMax`. Establishing these conventions should be straightforward. JSRs 181, 250, 303, and 305 have already walked this path for other domains; a proposal for WCET annotations would merely follow in their footsteps.

We intend to tackle the first issue by working with the JSR-308 Expert Group to expand the proposal to include loops and blocks, as Intel, Nortel, and others have suggested. If the JSR-308 proposal cannot be altered from its original form, we intend to submit a new proposal through the Java Community Process. (Now that Java is being released under an open-source license,[10] this effort should be much easier. For instance, we will be able create a reference implementation for this proposal by modifying the official `javac` compiler.)

[10] `http://www.sun.com/software/opensource/java/`

References

[1] G. Bernat, A. Burns, and A. Wellings. Portable worst-case execution time analysis using Java byte code. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (Euromicro-RTS 2000)*, pages 81–88, Los Alamitos, CA, USA, June 2000. IEEE Computer Society.

[2] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, January 2000.

[3] J. Consortium. Real-time core extensions, September 2000.

[4] J. Gosling, B. Joy, G. L. S. Jr., and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley Professional, Boston, Massachusetts, 3 edition, June 2005.

[5] E. Y.-S. Hu, G. Bernat, and A. Wellings. Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems. In *Proceedings of the Fifth IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2002)*, pages 109–116, Los Alamitos, CA, USA, April 2002. IEEE Computer Society.

[6] E. Y.-S. Hu, G. Bernat, and A. Wellings. A static timing analysis environment using Java architecture for safety critical real-time systems. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 77–84, Los Alamitos, CA, USA, January 2002. IEEE Computer Society.

[7] E. Y.-S. Hu, A. Wellings, and G. Bernat. Gain time reclaiming in high performance real-time Java systems. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*, pages 249–256, Los Alamitos, CA, USA, May 2003. IEEE Computer Society.

[8] E. Y.-S. Hu, A. Wellings, and G. Bernat. XRTJ: An extensible distributed high-integrity real-time Java environment. In *Proceedings of the Ninth International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003)*, volume 2968 of *Lecture Notes in Computer Science*, pages 208–228. Springer Berlin, February 2003.

[9] R. Kirner and P. Puschner. Discussion of misconceptions about WCET analysis. In *Proceedings of the Third International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, pages 61–64, July 2003.

[10] E. Kligerman and A. D. Stoyenko. Real-time Euclid: a language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[11] D. Lammers. REAL-TIME JAVA: Reliability quest fuels RT Java projects. *EE Times*, March 2005.

[12] M. M. Papi and M. D. Ernst. Annotations on Java types. `http://jcp.org/en/jsr/detail?id=308`, November 2006.

[13] P. Persson and G. Hedin. An interactive environment for real-time software development. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 2000)*, pages 57–68, Washington, DC, USA, June 2000. IEEE Computer Society.

[14] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the Fourth International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, October 2006.

[15] M. Schberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, Vienna, Austria, January 2005.

[16] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, pages 230–265. London Mathematical Society, November 1936.