Specification and Runtime Checking of Timing Constraints in Safety Critical Java

Ghaith Haddad

CS-TR-12-06
Oct 2012

Keywords: SafeJML, Safety Critical Java (SCJ), Java Modeling Language (JML), timing behavior, duration, performance, WCET.

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Department of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA

Specification and Runtime Checking of Timing Constraints in Safety Critical Java

by

Ghaith N. Haddad
B.S. Electrical Engineering, Jordan University of Science and Technology, 2001
M.S. Computer Engineering, University of Central Florida, 2007

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2012

Major Professor: Gary T. Leavens

ABSTRACT

The Java platform is becoming a vital tool for developing real-time and safety-critical systems. Design patterns and the availability of Java libraries, both provide solutions to many known problems. Furthermore, the object-oriented nature of Java simplifies modular development of real-time systems. However, limitations of Java as a programming language for real-time systems are a notable obstacle to producing safe real-time systems. These limitations are found in the unpredectible execution model of the language, due to Java's garbage collector, and the lack of support for non-functional specification and verification tools. In this dissertation I introduce SafeJML, a specification language for support of functional and non-functional specifications, based on an implementation of a safety-critical Java platform and the Java Modeling Language (JML). This dissertation concentrates on techniques that enable specification and dynamic checking of timing constraints for some important Java features, including methods and subtyping. SafeJML and these dynamic checking techniques allow modular specification and checking of safety-critical systems, including those that use object-orientation and design patterns. Such coding techniques could have maintenance benefits for real-time and safety-critical software.

ACKNOWLEDGMENTS

This work was supported in part by NSF grant CCF-0916350 titled "SHF: Specification and Verification of Safety Critical Java." The author also thanks Ales Plesk and Purdue team for their support and help.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Safety-critical systems are systems in which an incorrect response or an incorrectly timed response can result in significant loss to its users; in the most extreme case, loss of life may result from such failures. Safety-critical systems are usually also real-time systems. They can vary from simple single processor embedded systems with a few hundreds lines of code, up to a multi-language system with a few millions lines of code. They must also be extremely reliable and safe, as they control critical systems like chemical plants and nuclear reactors. Moreover, response time is crucial in these systems, as they interact with a hardware interface [1].

Software reliability is a major issue for safety-critical systems. Many life-threatening accidents have been caused by real-time software malfunctions or bugs. Examples include the Apollo 11 mission, where an overloaded processor nearly caused the first moon landing to fail, and the Therac-25 radiation therapy machine, where a timing bug in the code controlling the machine resulted in five patient deaths [2]. Many other examples can be found in literature and news [3]. For this reason, safety-critical applications require an exceedingly rigorous validation and certification process. Such processes are often required by legal statute or by certification authorities. For example, in the United States, the Federal Aviation Administration requires that safety-critical software be certified using the Software Considerations in Airborne Systems and Equipment Certification (DO-178B).

In order to achieve these certification requirements, higher-level programming languages and middleware are needed to robustly and productively design, implement, integrate, validate, and enforce both real-time constraints and conventional functional requirements, while assuring modularity and composability of independently-developed components. Such higher software productivity is important in the production of real-time embedded systems, given their intrinsically higher development costs.

However, tools and methodological support for embedded and real-time software development are lacking, especially compared to the case of standard software development. This is an important issue when embedded systems are tightly integrated with back-end applications and systems. Programmer productivity could increase if programmers were able to use the same development environment for both parts. Consider, for example, a monitoring project where one requirement is to place a device that reads some sensors, collects the data and sends it to a main server, where users can log in through a web interface and generate reports from the submitted data. A typical implementation for the system is to use Java to

develop the server-side and the web-report generator, but then for the development of the monitoring device software, developers have to use C, due to its support on such platforms. This will cause some code duplication, for example, the sensor reading can be stored in a Java class on the back end, but then it will be recorded using a different data structure in a different programming language at the sensor side, resulting in two different implementations of the same object on the two hardware platforms. This duplication is difficult to avoid[1] and may lead to inconsistencies that can manifest themselves as defects, and in any case causes problems for productivity and maintenance. Besides, programming and debugging in C takes, on average, more time than using Java[4].

The Java programming language has become an attractive choice because of its productivity, relatively low maintenance costs, and availability of trained developers. Although it has good software engineering characteristics, Java Standard Edition (Java SE) is unsuitable for real-time embedded systems due to underspecification of thread scheduling, synchronization, and garbage collection. This has led to the development of the Real-Time Specification for Java (RTSJ). RTSJ is a set of specifications and extension definitions for both the Java programming language and the Java Virtual Machine (JVM), oriented to enhancing language performance in real-time and embedded systems. RTSJ was used in many applications such as unmanned aircraft [5, 6, 7], real-time in circuit emulation [8], satellite applications [9], audio processing [10], industrial automation [11], and flight entertainment [12]. Furthermore, the Java community has also come up with Safety-Critical Java (SCJ), a variant of Java which is designed to enable the creation of safety-critical applications.

Deploying Java in safety-critical conditions requires advancing the state of the art in programming language implementation, specification and verification techniques. While there has been much prior work on proving the functional correctness of safety-critical systems, there has been less work combining notions of functional correctness with work on timing constraints. My goal is to work within the framework of SCJ, and to develop an integrated approach that permits modular specification and verification of functional and timing behavior for safety-critical real-time applications.

1.1 Problem Definition

The problem I aim to solve in this work is to design, evaluate and improve modular specification and checking tools for real-time and safety critical programs that are composed of Java and low-level systems code, based on an implementation of SCJ. In this section, two aspects of the problem that need to be solved are identified.

[1] One way to avoid this is by actually implementing this code in C and then use JNI to interface with the code at the server side. This is not ideal however, since the code is now implemented in C and the programmer has to deal with two different code bases, something we are trying to avoid in this work.

```
1  public class TransientDetectorScopeEntry {
2
3    public List createMotions() {
4      final List ret = new LinkedList();
5      Aircraft craft;
6      for (int i = 0, pos = 0; i < currentFrame.planeCnt; i++) {
7        // restrict maximum loop iterations
8        // perform some calculations...
9        if (old_pos == null) {
10         // new aircraft; calculate more...
11         //...
12       } else {
13         // old aircraft; other calculations...
14         //...
15       }
16
17       ret.add(m);
18     }
19     return ret;
20   }
21 }
```

Figure 1.1: The `createMotions()` method from the class `TransientDetectorScopeEntry` in the collision detection benchmark.

1.1.1 Problem I: Specifying Timing Constraints

1.1.1.1 Background and Related Work on Specification Languages for Timing Constraints

The first problem that needs to be solved is to take a specification language with adequate support for behavioral constraints, and extend it by adding support for non-behavioral constraints. For this purpose I extend the Java Modeling Language (JML) [13, 14, 15, 16, 17]. JML is a behavioral interface specification language [18] that boasts many state-of-the-art features for functional specification. However, the original design of JML had minimal support for specification of timing constraints, realized in its `duration` clause. This clause was based on the work of Krone *et al.* [19]. It is designed to specify the maximum time (i.e., worst case execution time or WCET) needed to process a method call in a particular specification case in units of "JVM cycles" [17].

JML's minimal support is not adequate for specifying complex timing constraints that can be used in state-of-the-art verification tools. To further describe the problem, consider

the example in Figure 1.1 on the preceding page. This example is taken from MiniCDj, a SCJ rewrite of the CDx[2] benchmark suite [20]. The example is a method call, `createMotions()` from the class `TransientDetectorScopeEntry`. This method computes the motions and the current positions of aircraft; it then stores the positions of these aircraft in a state table.

A standard methodology associated with such an example is for the programmer to be able to predict and analyze the time it takes for this method to complete, that is, the WCET. This can be done by performing duration measurements on the method of interest. In order to measure the method duration, the programmer can inject some code that records the start and end time of this method and calculate the duration of a call's execution. Another approach is to use a static analysis tool and provide it with enough information to be able to predict (or calculate) the WCET for a method[21, 22, 23]. While the first technique is easier for the programmer than the second, it is subject to errors as each programmer will have their own technique for performing these calculations. On the other hand, the second technique requires additional information to be provided to the tool in order for the tool to be able to accurately perform WCET analysis. This includes maximum number of iterations for loops and feasible paths.

1.1.1.2 Overview of SafeJML Approach to Specify Timing Constraints

This work is intended to provide tool support for timing analysis by providing a complete implementation of the `duration` clause with support for runtime checking. It is also intended to provide more timing specification constructs for JML that are not currently present, which makes SafeJML ready to support static analysis tools. For example, a static verification tool might need prior knowledge about the maximum number of times the loop in lines 6-18 can execute in order to verify the duration specification for that method. Such language support to specify the maximum loop iterations in the loop used in the example can significantly enhance the analysis. More details on various JML features that can be used to specify this case is presented in Section 2.5 on page 18.

Figure 1.2 on the following page shows another example; the method `theta()` from the class `VectorMath` from the same example benchmark. This method performs 3D vector math to calculate the angle between the X-axis and a vector in the 3D space. Similar to the first example, it is intuitive that paths 1 and 2, shown in lines 6 and 14 respectively, are together infeasible. This fact will affect the analysis because prior knowledge regarding the number of executable statements reflects directly on the method execution time. Hence, language support to state that some set of paths are mutually infeasible could help improve timing calculations.

[2] The CDx benchmark suite is an open-source family of collision detection benchmarks[20].

```
final class VectorMath {

  public static float theta(Vector3d a) {
    float x = a.x, y = a.y;
    if (x == 0) { // tangent undefined for x = 0
      // path 1
      if (y == 0) throw new ZeroVectorException("undefined");
      if (y < 0) return (float) (1.5 * Math.PI);
      return (float) (0.5 * Math.PI);
    }
    float t = (float) Math.atan(y / x); // calculate theta

    if (x < 0) {
      // path 2
      // path 1 and path 2 together are infeasible
      return (float) Math.PI - t; // adjust quadrant
    }
    if (t < 0) t += 2 * Math.PI; // range adjustment [0, 2*pi]

    return t;
  }
}
```

Figure 1.2: The `theta()` method from the class `VectorMath` in the collision detection benchmark.

1.1.2 Problem II: Specification and Runtime Checking Techniques

The second problem that is identified for this work is to provide specification and runtime checking techniques to be able to specify and check the correctness of SCJ programs with SafeJML annotations. One major issue that we address in this work is subtyping[3].

1.1.2.1 Background

Subtyping is an important coding technique for Object-Oriented (OO) programming. However, real-time programmers usually try to avoid subtyping [25], due to the perceived complications for timing analysis. This paper describes both specification and verification and runtime checking techniques that handle these complications.

Consider the example in Figure 1.3 on the next page and Figure 1.4 on page 8. Figure 1.3 declares a type `Vector2d` that represents two-dimensional vectors. Figure 1.4 shows another type, `Vector3d`, which is a subtype of `Vector2d`. In these figures a representative method in these types is shown, the `scale` method, which scales the dimensions of the vector by a floating point number `factor`.

As can be seen in Figure 1.3, SafeJML takes from JML many features for specification of functional behavior. (These include a `requires` clause, which specifies a method's precondition, an `assignable` clause, which specifies what locations a method is allowed to assign, and an `ensures` clause, which specifies a method's postcondition.) The `duration` clause specifies the method's maximum time using some constants that depend on the platform for which this code is written.[4]

Another important feature in this example is what JML calls a "specification case" [16]. In Figure 1.3 on the next page, the method's only specification case starts with the keywords `public normal_behavior` and consists of a requires clause followed by a group of other clauses. The requires clause governs when the specification case is enforced; whenever the precondition is satisfied by the method's arguments, then the rest of the clauses in the specification case must also be satisfied by the method's execution. (The `normal_behavior` keyword means that when the specification case's precondition is satisfied, the method cannot throw an exception, but must return normally.)

As in JML, SafeJML specification cases are inherited by overriding methods in subtypes. Thus whenever the specification case for an overridden method in a supertype has a precondition that holds, that entire specification case, including any duration clauses, must be obeyed by the subtype's overriding method.

[3]Some text of this section is adapted from our paper "Specifying Subtypes in SCJ Programs" [24].

[4] These platform dependencies cause no conceptual problems, because they are statically-known constants.

```
1  public class Vector2d {
2
3      //@ public model JMLDatagroup dims;
4      protected /*@ spec_public @*/ float x, y; //@ in dims;
5
6    /*@   public normal_behavior
7      @     requires !Float.isNaN(factor);
8      @     assignable dims;
9      @     ensures x == \old(x) * factor && y == \old(y) * factor;
10     @     duration 2 * (ITimeConstants.MultiplyTime
11     @                   + ITimeConstants.AssignTime);
12     @*/
13     public void scale(float factor) {
14         this.x *= factor;
15         this.y *= factor;
16     }
17 }
```

Figure 1.3: An Example Supertype in SCJ with SafeJML specifications. In SafeJML special comments that start with an at-sign (of form `//@` or `/*@`) and in which at-signs are ignored at the beginnings of lines, are parsed by the SafeJML checker. Other methods of this type are omitted.

The subtype, `Vector3d` shown in Figure 1.4 has an extra field, `z`, that records the additional dimension. This is typical of OO programs, as subtype objects often hold more information than objects of the corresponding supertypes. The specification of `scale` in this subtype `Vector3d` can be thought of as having two specification cases: one inherited from the supertype and the one written in Figure 1.4. Each specification case must be obeyed when its precondition is true. The keyword `also` in the subtype's specification reminds the reader that there are other specification cases inherited from the supertype.

SafeJML uses specification inheritance because it automatically makes each subtype a behavioral subtype of its supertypes [16, 26, 27]. Behavioral subtyping is necessary for the validity of supertype abstraction, which is the standard methodology for modular reasoning in the presence of subtype polymorphism. *Supertype abstraction* uses the specifications associated with each receiver's static type to reason about method calls [16, 27, 28]. For example, if `v2` has static type `Vector2d`, then using supertype abstraction, a call such as `v2.scale(4.5)` would be guaranteed to take at most the time specified for `Vector2d`'s `scale` method in Figure 1.3. Supertype abstraction says that this guarantee is valid even if `v2` denotes an object whose runtime type is `Vector3d`.

However, applying specification inheritance and behavioral subtyping to timing constraints poses a practical problem. As in the aforementioned example, it is often the case

```
1  public class Vector3d extends Vector2d {
2
3      protected /*@ spec_public @*/ float z; //@ in dims;
4
5    /*@ also
6      @   public normal_behavior
7      @     requires !Float.isNaN(factor);
8      @     assignable dims;
9      @     ensures z == \old(z) * factor;
10     @*/
11     public void scale(float factor) {
12         super.scale(factor);
13         this.z *= factor;
14     }
15 }
```

Figure 1.4: `Vector3d`, a subtype of `Vector2d`. The specification of its `scale` method adds to the specification inherited from its supertype, `Vector2d`. Other methods of this type are omitted.

that the subtype's instances contain more information than those of its supertypes. Now consider the timing constraints on the `scale` method of both types. This method has a very tight specification in `Vector2d`. That specification only permits just enough time for a reasonable implementation. However, the override of this method in the subtype `Vector3d`, as mentioned, must obey the specification given in `Vector2d`, and thus it is only allowed time to do two multiplications and assignments. However, `Vector3d` will not be able to scale the vector within the time specified in the inherited duration clause.[5]

1.1.2.2 Related Work on Runtime Checking Techniques for Subtypes

One can try to avoid such problems with behavioral subtyping by using several techniques. One technique is to not override methods in subtypes, for example, by giving them a different name [29]. (This is equivalent to declaring all methods to be `final`.) However, this technique precludes the use of subtype polymorphism, which obviates all the maintenance advantages of typical OO design (such as standard OO design patterns).

[5] Thus as specified so far, `Vector3d` cannot be correctly implemented. This is the way that problems with behavioral subtyping manifest themselves in SafeJML, due to SafeJML's use of specification inheritance.

Another technique for avoiding overly strict constraints on subtype methods is to use underspecification; that is, one weakens the supertype's specification enough to allow the subtype's implementation to satisfy the supertype's specification. For example, if the specification in Figure 1.3 is changed such that the duration expression was multiplied by 3 instead of 2, then this would allow the implementation in `Vector3d` to be correct. However, this technique will make reasoning about timing constraints imprecise, since each supertype's specification will necessarily have to be loose enough to allow the slowest subtype's implementation. Further, this technique causes maintenance problems, as adding a new, slower, subtype will require changing the specifications of all the methods it overrides (as described for `Vector2d`). That might require reverification of client code that was previously verified, to take this new, weaker, specification, into account.

Finally, neither of these techniques can be applied to programs that use some prespecified OO libraries, if clients need to subtype types in the library override its methods (as happens in OO frameworks).

1.1.2.3 Overview of SafeJML Approach of Runtime Checking Techniques for Subtypes

In this work, I solve the problem that the constraints of behavioral subtyping impose on real-time software by designing a solution that both maintains the flexibility of OO programming and still makes supertype abstraction a valid reasoning principle, so that reasoning about timing constraints is modular. I also introduce a tool implemented to experiment with these ideas and demonstrate that the proposed approach is feasible.

1.2 Contribution

The major focus of this work is on specification and checking of real-time systems written in Java. I extend previous research on functional runtime checking using JML to non-functional timing properties and show a way to perform timing analysis for the program compiled and executed on the SCJ VM, namely a large C program that integrates both the original Java code as well as device drivers and other components written in C.

I provide, for SCJ methods (including native methods), a modular specification and runtime checking technique for tight timing constraints. This technique is to be embodied in an extension of JML tailored to SCJ programs. Furthermore, introduced specification techniques could be used to build modular verification tools and techniques that are both

sound and static. Although the original aim was to design this verification tool, this was left to future work.

In order to achieve my goal, my solution has the following requirements:

Expressiveness. The specification language should be useful for expressing specifications of timing constraints for safety-critical and real-time systems [30]. Expressiveness will be evaluated by a case study.

Precision. The tool's analysis of worst case execution times must also be precise enough to be useful. There is no specific standard to compare the precision of certain analysis against, therefore, acceptable precision can only be domain dependent. For example, while a program that controls a plane requires precisions up to the millisecond level, a program that controls a sprinklers system in a farm might consider timing properties rounded to the second level precise enough. In other words, precision is a measure that can be only gauged by the user. Precision will be evaluated experimentally.

Direct feedback. Error messages provided by the tools regarding violations of timing constraints must relate directly to the smallest specified units (e.g., methods or individual blocks of source code) where timing constraints might be violated, and must provide specific information about those (potential) violations. This requirement will be validated analytically using a case study.

Efficiency. Tools developed should be modular and efficient enough for developers to use interactively. Modularity will be validated analytically, while the efficiency of the tools will be evaluated using measurements on a case study.

The technical goals of this project partially depend on the availability of a Java virtual machine that is suitable for use in the context of automated verification of functional and timing properties of programs. Today most Java virtual machines, and therefore, the applications designed to run using these virtual machines, are not amenable to verification, as they are written in a mixture of high-level code (Java), system code (C) and hand tuned assembly with key algorithms tuned for throughput rather than analyzability.

In order to achieve this goal, part of this work has been conducted by a our partner group at Purdue. That group focuses on the design and implementation of FijiVM, a Java virtual machine that supports the Safety-Critical Java (JSR-302) API, and is designed from the start so that the applications running on top of it are analyzable. This virtual machine is written entirely in Java and will self-compile to portable C code.

The following is an overview of each chapter.

Chapter 2 In this chapter I introduce some background and essential definitions for the reader.

Chapter 3 In this chapter I introduce the SafeJML specification language, describe its details, and introduce some examples.

Chapter 4 In this chapter I introduce a novel approach to specify subtypes in SCJ programs using SafeJML, I describe the details of this approach and show by example how this approach can be implemented.

Chapter 5 In this chapter I show the results of an evaluation to the system introduced throughout this work against the proposed solution requirements presented earlier. The evaluation is done on examples introduced earlier in literature.

Chapter 6 This chapter concludes the dissertation. I summarize the limitations of the current implementation and future work, and I list the contributions of this work.

CHAPTER 2
BACKGROUND

This chapter gives some background and definitions that are essential for this work. The main idea behind the software engineering discipline, and some basic concepts about the Java programming language and its extension to support both real-time and safety critical programming, RTSJ and SCJ, are introduced. A brief introduction about the specification language of interest, JML, and the tools that supports this language to perform static and dynamic verification is also given.

2.1 Java Programming Language

Java was designed in 1991 as a programming language for the consumer electronics industry. It gained more attention after the Internet began to spread in 1994. Gosling and McGilton, the authors of the Java programming language, defined the advantages of Java in their famous white paper [31] as *a*) "Simple, Object Oriented, and Familiar." *b*) "Robust and Secure." *c*) "Architecture Neutral and Portable." *d*) "High Performance." *e*) "Interpreted, Threaded, and Dynamic." Many of those same advantages serve as serious limitations to Java support for real-time and embedded systems. Currently, compiled languages such as C and C++ are the choice to develop large-scale high-performance real-time and embedded systems. Some reasons for choosing C and C++ are their large support in embedded systems and their small runtime footprint, which is a desired property when dealing with small micro-controllers. However, when developed using languages such as C and C++, these systems might have unreliable memory management due to pointer arithmetic operations, or difficult implementation and debugging. Other problems in C and C++ include the lack of native threading support, and having to maintain different binary distributions for different architectures: drawbacks that were all targeted by design in Java. On the other hand, Java's high resource requirements for the JVM, unpredictable real-time behavior, and underspecification of thread scheduling, synchronization, and garbage collection, make standard Java unsuitable for real-time applications.

This has lead to the development of the Real-Time Specification for Java (RTSJ) [32]. RTSJ represents a standardized approach to enhancing the Java virtual machine by tightening the semantics of thread scheduling and synchronization and providing mechanisms that allow real-time Java programs to run without interference from garbage collection. The

RTSJ has enabled development of large-scale real-time embedded systems in the Java language [33, 34, 35]. Real-time Java has established itself as a viable alternative for developing large real-time systems as evidenced by commercial virtual machines such as Mackinac [36], WebSphere Real Time [37], PERC [38], and JamaicaVM [39]. Recently, Lockheed Martin used the PERC Ultra, a Java VM with predictable garbage collection, to modernize the Aegis cruiser fleet [40], while IBM and Raytheon have teamed to build the battleship computing environment software for the new DDG-1000 warship [41]. Figure 2.1 shows a sample Real-time Java application with Purdue's Ovm real-time JVM: the first integrated real-time Java system on the ScanEagle Unmanned Aerial Vehicle [5]. The flight hardware/software was an Embedded Planet PowerPC 8260 processor running at 300MHz with 256Mb SDRAM and 32 Mb FLASH. The operating system is Embedded Linux.

Figure 2.1: A Sample Real-time Java application. Credit: Jan Vitek

Where the RTSJ strives for expressiveness and imposes few limitations on how a developer structures an application in terms of concurrency, packaging, synchronization, memory, etc., safety-critical applications must conform to rigorous certification requirements, and use much simpler programming models that are amenable to certification. For this reason, the Java community created SCJ. SCJ was developed as a Java Specification Request (JSR-302), and is designed to enable the creation of safety-critical applications built using a safety-critical Java infrastructure and using safety-critical libraries that are amenable to certification under DO-178B, Level A and other safety-critical standards.

2.2 Software Engineering and Formal Methods

Software Engineering is the discipline aimed at producing software with a minimum number of faults, delivered on time and within budget, that satisfies a client's needs [42]. This has become even more important in certain areas of software development. For example, in safety-critical systems, software safety is a key issue in the development process. In order to determine whether a certain software is safe, one thing developers and testers usually do is test the software for correctness against its requirements. However, even with testing, there is no guarantee that the program will satisfy all requirements decided on in the requirements analysis phase. Besides, these tests are usually intended to eliminate the functional errors in the software, that is, whether the program delivers the correct results or not. In addition to functional requirements, real-time programs must satisfy timing constraints [43, Chapter 12]. Both time and space requirements affect the life-cycle of program development. This is due to the additional coding and tests the programmer has to write and perform to satisfy these constraints.

One way to guarantee that the overall implementation of the system is correct, is to use *formal methods*. The term "formal methods," refers to the mathematical tools and techniques that are used to ensure correctness. A requirements document can be written using statements in mathematical logic. The requirements are then refined to construct a formal, mathematical description of the system that can be used to symbolically test the entire state space of the design, and formally verify the correctness of the implementation.

Formal specifications are a promising approach for the design, documentation, verification, debugging, and testing of software systems [44, 43, 45]. Specifications are important to achieve verification modularity, and modularizing a program's proof of correctness can greatly simplify the verification task of that program, as the verification task is now simplified to focus on the specifications of the method being verified. This means that if the implementation of that method changes and its specifications remain the same, it is not required to verify calls to the method again. This property is essential for scalability of verification; without specifications, verification algorithms can introduce a state space that is very large in size, which makes the verification process both time and resource consuming. In other words, it is desired that the introduced complexity of an extended system to be verified with only an amount of work proportional to the size of the additional part added to the system.

2.3 Timing Analysis

Timing analysis forms the foundation for this work. The basic problem for any real-time designer is to ensure that the designed tasks are completed before (or no earlier than)

a certain deadline. This can be done by performing timing analysis on the implementation. The two major methods to perform analysis are:

Dynamic Analysis (or measurements). The idea is to measure the time that a task takes at run-time. Typically this is done using tools that instrument either the source code or the binaries with code to measure execution time. If we eliminate the underlying features in any hardware that causes the timing characteristics to be nondeterministic, such as caches, then this could be a very precise method for tasks with no user input or for tasks that have very clear and small sets of input parameters. However, in many cases, those underlying features cannot be eliminated. Furthermore, if the measurements were performed using an incomplete set of inputs, which will usually be the case, or if timing analysis is to be performed on a software library for which inputs are unknown, then dynamic analysis will not be able to measure the worst case timing results for all inputs. Thus, in most practical cases, dynamic analysis cannot give worst case execution time bounds.

Static Analysis In contrast with dynamic analysis, static analysis uses calculations rather than measurements to extract a task execution time. In this case, the analysis can extract the desired results without the need for a complete set of tests that can cover all possible execution paths. Instead, path coverage and duration analysis is performed on the code itself or on the compiled binaries without the need for the target hardware to execute the code.

Some of the features of static analysis, while not necessary for the analysis itself, are very important if it is desired to have precise analysis, or tight timing bounds, by eliminating the false positives. Two major issues for obtaining tight timing bounds are *path sensitivity* and *context sensitivity*. We say that an analysis is *path-sensitive*, if it depends on the predicates at the conditional statements to consolidate execution times for the different branches formed by these conditional statements. While path insensitive analysis can still do the job, it is usually more conservative (i.e., safer), but the result may contain more imprecise bounds.

The second feature of static analysis that is important is its context sensitivity. A context is simply the range of values a parameter or environmentally-bound variable can take. A static analysis method is *context-dependent* if it performs different analysis for method calls depending on the place in the code where the method is called. Differing contexts can highly affect the execution times of any system, especially library code. For the timing analysis to be tight, it is important for the system that performs the analysis to be both path-sensitive and context-sensitive.

2.4 Safety Critical Java (SCJ)

While Java offers the possibility of high software productivity, it is not tailored for safety critical or real-time systems. In particular, Java's garbage collection may cause unpredictable worst case execution times. To avoid these problems, SCJ was developed as a Java Specification Request (JSR-302) [46]. SCJ is designed to enable the creation of safety-critical applications built using a safety-critical Java infrastructure and using safety-critical libraries that are amenable to certification under DO-178B, Level A [47] and other safety-critical standards. JSR-302 is near completion now [48].

Prior to providing an implementation for SCJ, Vitek's group at Purdue University implemented first the Open Virtual Machine (Ovm). The Ovm is a generic framework for building virtual machines with different features. It supports components that provide a wide variety of alternate implementations of core VM features. While Ovm's internal interfaces have been carefully designed for generality, much of the coding effort has focused on implementations that achieve high runtime performance and good predictability with low development costs. The real-time support in Ovm is compliant with version 1.0 of the RTSJ. Sources and documentation for Ovm are available under an open source license [49], and further discussion can be found in various papers by its designers [5, 50, 51].

The current implementation of Ovm relies on an optimizing compiler that translates the entire application and virtual machine code into C which is then processed by the Gnu C Compiler (gcc).

Vitek's group has also produced oSCJ [52], an open-source SCJ implementation based on OVM [49]. (Currently oSCJ implements all of Level 0 of SCJ [46], which specifies three different levels to implement, Level 0 through 2). Like OVM, oSCJ takes the approach of compiling both the SCJ code and virtual machine into a (large) C program, which is then compiled with a standard C compiler (such as gcc) appropriate for the hardware.

SCJ programs use the concept of the class `Mission`, which in essence is a certain task that needs to be executed according to a certain schedule, which defines the scheduling constraints of the mission, such as release time, deadline, and priority. Schedules use event handlers to restrict the execution of the mission.

In SCJ Level 0, the class `PeriodicEventHandler`, is the only way to establish a periodic activity, and thus permits the automatic execution of SCJ code. The constructor for `PeriodicEventHandler` specifies an object of type `PeriodicParameters`, which in turn is a subtype of `ReleaseParameters`. SCJ specifications for these classes are shown in Figure 2.2. In SCJ, the object `PeriodicParameters` specifies the periodic release parameters for a specific periodic activity of type `PeriodicEventHandler`. These specifications are different for different levels of SCJ. For example, in SCJ Level 0, only the start time and mission period are required, while in Level 1, a deadline that defaults to the period and a miss handler are specified. In general, timing analysis for SCJ is meant to be performed offline, and only in Levels 1 and 2 is deadline miss detection supported dynamically.

```
1  @SCJAllowed
2  @SCJRestricted({INITIALIZATION})
3  public PeriodicEventHandler(PriorityParameters priority,
       PeriodicParameters release,
4  StorageParameters storage)
5
6  @SCJAllowed
7  public abstract class ReleaseParameters implements Cloneable
8
9  @SCJAllowed
10 protected ReleaseParameters()
11
12 @SCJAllowed(LEVEL_1)
13 protected ReleaseParameters(RelativeTime deadline,
       AsyncEventHandler missHandler)
14
15 @SCJAllowed
16 public class PeriodicParameters extends ReleaseParameters
17
18 @SCJAllowed
19 public PeriodicParameters(HighResolutionTime start, RelativeTime
       period)
20
21 @SCJAllowed(LEVEL_1)
22 public PeriodicParameters(HighResolutionTime start, RelativeTime
       period
23 RelativeTime deadline, AsyncEventHandler missHandler)
```

Figure 2.2: SCJ Levels 0 and 1 Specifications for `PeriodicEventHandler`, `ReleaseParameters` and `PeriodicParameters`

The worst case execution time needed to execute a schedulable object is meant to be provided by the SCJ's release parameters. Level 0 specifications barely support that, especially with no means to handle deadline misses. On the other hand, Level 1 supports miss handlers only at the level of the schedulable object (or the `PeriodicEventHandler`). This leaves little room for the programmer to specify duration at the method level. SafeJML allows specification of durations and thus deadlines at the method level, this provides more detailed specification and verification of timing constraints.

2.5 JML - The Java Modeling Language

Design by Contract (DBC) is a formal specification approach that uses the idea of assertions for specifying code using the same language that the executable code is written in [53]. In DBC for object-oriented language like Java, classes are seen as clients of other classes. Thus, a client calling a method must have a "contract" with the class whose method is being called. Prior to calling that method, the client must guarantee certain conditions, and in return, the called method must guarantee certain conditions to hold when the method has completed execution [54]. These contracts provide formal documentation for the code. DBC took this idea one step further by introducing keywords and syntax for these contracts, and make them executable by the machine that is running the compiled code [53], thus introducing an environment for formal verification or runtime checking of the code, so any violation of these contracts can be detected and reported.

The Java Modeling Language (JML) [13, 55, 14, 15, 56, 57, 16] is a behavioral interface specification language (BISL) [18] that follows the Design by Contract approach to specify Java modules [58, 59]. JML blended ideas from Eiffel [60, 61, 53], Larch [18, 62, 44], and the refinement calculus [63, 64, 65, 66] in its approaches to specification. It adds many state-of-the-art features for functional specification. Readers are referred to the *JML Reference Manual* [67] for more information.

Many tools have been developed to support JML, and to provide runtime and static checking to verify functional requirements. Burdy *et al.* [13] list such tools. For example, `jmlc` [68] is used for runtime assertion checking. `ESC/Java` [69], `ESC/Java2` [70], `JACK` [71], and `LOOP` [72] are all used for static checking and verification.

JML provides minimal support for specification of timing constraints, realized by using the **duration** clause [73], based on the work of Krone *et al.* [19]. This clause is designed to specify the maximum time (i.e., worst case execution time) needed to process a method call in a particular specification case in "JVM cycles".

JML's duration clause has never before been implemented in the tools or used in actual case studies. In addition, JML lacks the means to write timing constraints using intervals. Many real-time systems consider executing tasks no earlier than certain times.

A good example is the Pacemaker System Specification [74] from Boston Scientific. The pacemaker system specifies events that must occur "not later than" a certain time, and also "not earlier than" a certain time. The current duration clause in JML makes it difficult to specify systems like this that need interval-based specification of timing constraints.

Another limitation in JML is the limited ability to specify bounds on loops. The only clause that can be used to annotate loop bounds is the **decreasing** clause. This clause specifies a loop variant function using an expression, of type integer or long, that must be decreased by at least one each time the loop executes. This clause is mainly used to specify loop termination. Furthermore, as the variant is always decreasing, it is guaranteed that the variant represents an upper bound on the number of times the loop executes.

However, the **decreasing** clause does not directly address the concept of maximum loop iterations. Furthermore, the definition of the loop variant function can be either imprecise or hard to construct if used to denote the maximum loop iterations. Consider the example in Figure 2.3 on the next page, in this example, the loop variant function `i` is decreasing, but the upper limit on the number of loop iterations is a constant value, which is the maximum number of bits in an integer (32 bits). This method can be annotated by introducing a ghost variable that decreases while the method executes[1]. Alternatively, the programmer can specify the loop with a constant maximum loop iterations, which is 32 in this case.[2] For that reason, a more direct construct is sought, one that involves less inference than the decreasing clause.

[1]The decreasing clause can be constructed from the variable `i` using a model field and assigning to it the formula `Math.round(Math.log(i)/Math.log(2))`.

[2]A better specification statement is to use the formula in the previous footnote.

```
1  public class exDecreasing {
2
3    /*@  public normal_behavior
4      @    requires i>=0;
5      @*/
6    public static int countbits(int i) {
7      int res = 0;
8      //@ decreasing i;
9      while(i>0){
10       i=i/2;
11       res++;
12     }
13     return res;
14   }
15   public static void main(String args[]){
16       System.out.println(countbits(2136));
17   }
18 }
```

Figure 2.3: Using the `decreasing` loop variant function to annotate a method that calculate the number of bits in an integer

CHAPTER 3
SAFEJML DESIGN AND IMPLEMENTATION

In this chapter[1], I introduce my approach for the design and implementation of the SafeJML Specification Language for SCJ programs. This approach solves the problem of improving modular specification and verification tools for real-time and SCJ programs that are composed of Java and low-level system code.

The proposed approach consists of three parts. The first part focus on the overall system architecture, while the second part focuses on the design and development of the SafeJML specification language. The third part discusses the design and implementation of a runtime assertion checker that utilizes SafeJML to check specifications at runtime. Chapter 4 is a continuation to this chapter. It is focused on solving the problem of timing constraints and behavioral subtyping.

3.1 Overall System Architecture

Several architectural decisions were made in order to be able to solve the problem that was identified in Chapter 1. The first decision was to extend an existing specification language (JML), and augment it with constructs to enable nonbehavioral specifications. The second decision was to integrate with current state of the art tools and compilers to solve this problem. Using these tools allowed to focus on the key challenges of the problem and get good results, without devoting a considerable amount of time to duplicating the effort that has gone into their development.

The overall architecture I propose is shown in Figure 3.1 on the following page. As shown in the figure, the analysis process consists of four stages:

SafeJML Compiler The first stage is the SafeJML compiler. The SafeJML compiler is a SCJ compiler with the ability to process SafeJML specifications. The input for this stage is the SCJ code that needs to be analyzed along with the SafeJML specifications. The duration specifications are usually embedded inside the SCJ code itself. Alternatively, the SafeJML compiler, allows for specifications to be stored in separate (`.jml`)

[1]This chapter is based on our paper "The design of SafeJML, a specification language for SCJ with support for WCET specification" [75].

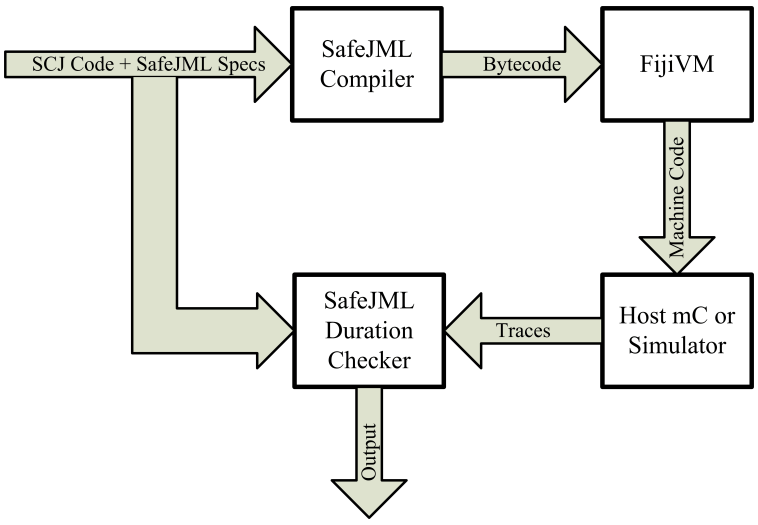

Figure 3.1: System Diagram

files. The output for this stage is the compiled class files (bytecode), which in turn is the input for the next stage, the Fiji compiler.

FijiVM Compiler During the tool's second stage, the Fiji compiler compiles the program's class files into C code which in turn is compiled into machine code. Then the generated machine code moves to the third stage, the host microcontroller.

The Host Microcontroller During the tool's third stage, the program's code is executed. This will result in producing execution traces. At the end of the program's execution, the analysis flushes the trace information into a trace file. This file will be used later to execute the verification part of the algorithm.

SafeJML Checker In the tool's fourth stage, execution traces are processed by the SafeJML checker, which also takes the program's code and specifications as an input. This stage analyzes the trace files and compares them to the specifications. The output of this stage contains warnings regarding any potential violation of the specifications. It uses the compiler output to produce reports of specification violations.

SafeJML was originally designed [75] to be used with both static analysis tools such as AbsInt's aiT [21, 22, 23] and dynamic analysis tools such as RapiTime [76] Static analysis is conservative and sound, which guarantees accuracy but may give less precise results than dynamic analysis. Dynamic analysis, on the other hand, is more precise, but requires test cases and provides no guarantees about all possible executions. Ernst has emphasized the importance of applying a combination of both static and dynamic analysis [77]. Thus SafeJML is designed to support both kinds of analysis.

In this work, I focus on dynamic analysis (runtime checking) for timing constraints. However, when I initially tried to use RapiTime for dynamic analysis, I discovered some

limitations (as of this writing) on the size of C files it can process in a given run. This limitation caused problems, because of the way the tool compiles SCJ to C code. Compilation of SCJ code takes place in a SCJ implementation, mainly using the oSCJ [52, 78] or the Fiji VM [79]. However, both of these VMs produce large C source files[2], and that caused RapiTime to stop functioning.

To overcome the problems with RapiTime, I implemented a simplified analysis tool that calculates an estimation of the worst case execution times for the specified methods, then compares the calculated values with the specifications of these methods. This procedure has enabled us to do simple dynamic checking. This tool is implemented as part of the SafeJML Runtime Assertion Checker, which is discussed in Section 3.3.1 on page 36.

Since SafeJML can also support static analysis, this work can be extended to develop static verification tools. However, implementing static analysis is left as future work.

3.2 The Design of SafeJML

Earlier in Section 2.5 on page 18, JML was introduced as a behavioral interface specification language for Java. In this section I focus on extending the JML language by adding support for timing constraints, which in the original design of JML appears in the form of the `duration` clause. This clause is designed to specify the maximum time (i.e., worst case execution time or WCET) needed to process a method call in a particular specification case in nanoseconds[3].

My design of SafeJML revises the `duration` clause to make it based on absolute time units, and adds several other features that enable the use of various tools to check timing constraints. The rationale for the language design and the initial implementation are also introduced. This is the first publicly released and documented real-time extension to JML.

3.2.1 An Overview of SafeJML

The SafeJML implementation has been developed as an extension to JAJML[4]. It is designed to allow SCJ users to specify both functional and timing behavior on a per-method basis.

[2] In a single run of oSCJ, the tool produced a 20MB C code file, which RapiTime tool could not handle.

[3] In order to understand the rest of this chapter, it is recommended to refer to the JML Reference Manual [17] for more information about JML design and usage.

[4] Please refer to Section A on page 71 for a detailed discussion on the construction of SafeJML as an extension to JAJML.

Specification of timing behavior on a per-method basis allows one to quickly isolate methods that exceed their time budget, which can speed debugging of (possibly rare) runs of the tasks that use these methods. The drawback of this feature is that programmers will have to spend effort dividing the time budget among the methods called in a task, and then recording this division of the time budget as timing specifications for each method. While this seems to be contrary to the way that real-time programmers typically work, we note that such SafeJML specifications can be added to methods in layers (first to the methods called directly from a task (or mission), and only later to methods called by those methods, if there is a problem). Furthermore, method timing specifications can also be added after the code is written and some measurements have been made, with the goal of catching executions that go over the expected timing budget. Finally, just as with functionality specifications, timing specifications can serve as contracts, and thus can support division of labor and allow modular reasoning about timing behavior.

As mentioned in Chapter 1, SCJ was developed as a Java Specification Request (JSR-302) [46]. SCJ was designed to enable the creation of safety-critical applications built using a safety-critical Java infrastructure and using safety-critical libraries that are amenable to certification under DO-178B, Level A [47] and other safety-critical standards. JSR-302 is near completion now [48]. Refer to Section 2.4 on page 16 for more information about SCJ.

3.2.2 A First SafeJML Example

The first example of SafeJML is taken from MiniCDj and is shown in Figure 3.2 on the next page. This example was first shown in Figure 1.1 on page 3.

To improve timing analysis for verification tools that use various annotations to restrict loop bounds, and restrict execution times, and prune infeasible paths. SafeJML uses annotation comments (which look like `/*@ ... @*/` or lines starting with `//@`), like JML's syntax, to specify SCJ programs. For example, in Figure 3.2, line 7 specifies the maximum number of times the loop may iterate. Line 10 specifies the maximum number of new airplanes that can be handled in a single frame. This restricts the number of times a new airplane will appear in a single frame.

Another example of SafeJML is taken from MiniCDj and is shown in Figure 3.3 on page 26. This example was first shown in Figure 1.2 on page 5. Similarly, lines 6 and 14 specify that the two paths in which these lines are included are mutually exclusive.

A key requirement in the implementation of SafeJML is to compile these annotations to the corresponding tool annotations to be used later during analysis.

```
1  public class TransientDetectorScopeEntry {
2
3    public List createMotions() {
4      final List ret = new LinkedList();
5      Aircraft craft;
6      /*@ maximum_loop_iterations MAX_PLANES_PER_FRAME; @*/
7      for (int i = 0, pos = 0; i < currentFrame.planeCnt; i++) {
8        // restrict maximum loop iterations
9        // perform some calculations...
10       /*@ local_worst_case MAX_NEW_PLANES_PER_FRAME; @*/
11       if (old_pos == null) {
12         // new aircraft; calculate more...
13       } else {
14         // old aircraft; other calculations...
15       }
16       ret.add(m);
17     }
18     return ret;
19   }
20 }
```

Figure 3.2: The `createMotions` method from the collision detection benchmark, with SafeJML annotations added.

3.2.3 SafeJML Syntax and Semantics

In this section we describe the syntax and semantics of the additions that SafeJML makes to JML.

3.2.3.1 Contract Clauses

SafeJML, like JML, specifies methods using contracts ("specification cases") [17]. As we will discuss below, the clauses in such a contract can also be used to specify the behavior of blocks of code. The clauses of particular interest here are the splits-wcet-clause and the duration-caluse, as shown in the grammar below.

simple-spec-body-clause ::= ... | *splits-wcet-clause* | *duration-clause*

```
1  final class VectorMath {
2
3    public static float theta(Vector3d a) {
4      float x = a.x, y = a.y;
5      if (x == 0) { // tangent undefined for x = 0
6        //@ path xeqzero \exclude xltzero;
7        if (y == 0) throw new ZeroVectorException("undefined");
8        if (y < 0) return (float) (1.5 * Math.PI);
9        return (float) (0.5 * Math.PI);
10     }
11     float t = (float) Math.atan(y / x); // calculate theta
12
13     if (x < 0) {
14       /*@ path xltzero \exclude xeqzero; @*/
15       return (float) Math.PI - t; // adjust quadrant
16     }
17     if (t < 0) t += 2 * Math.PI; // range adjustment [0, 2*pi]
18
19     return t;
20   }
21 }
```

Figure 3.3: The `theta` method from the collision detection benchmark, with SafeJML annotations added.

---

duration-clause ::= `duration` *spec-expression*`;`
spec-expression ::= ... | *interval-expr*
interval-expr ::= *spec-expression* .. *spec-expression*

Figure 3.4: Syntax of the `duration` clause.

Duration Annotations for Methods SafeJML, like JML itself [17], has a `duration` clause, that is intended for specifying the worst case execution time of a method (or block of code). The syntax of the `duration` clause is shown in Figure 3.4.

A *spec-expression* is an expression without side effects, which may use SafeJML's specification-only features. The *interval-expressions* can be used to specify a minimum, as well as a maximum, number of nanoseconds that the method's execution may take. If only one number is given, it is taken as the maximum number of nanoseconds allowed (this is equivalent to having the minimum number as 0).

The semantics of SafeJML's duration clause is based on the work of Krone *et al.* [19, 80]. Its minimum and maximum allowed for the method's execution time is given in nanoseconds.[5] We assume that the built-in SafeJML package named `org.jmlspecs.lang` contains definitions of constants such as `MS`, `SEC`, etc. to allow expression of timing constraints in units that are more convenient for the specifier. Figure 3.5 shows these definitions.

The execution time of a method call is the time from the end of parameter evaluation before the method is called, until the method returns to the caller.

As in JML, SafeJML method specifications can contain multiple specification cases (separated by the keyword `also`), each of which specifies the behavior when a certain precondition is met. When the precondition for more than one specification case holds, then each

[5] However, in SafeJML (as in JML) such a specification only applies when the precondition of the specification case in which it appears is satisfied [17, 16].

```
public static final long SEC = 1000000000;
public static final long MILLISEC = 1000000;
public static final long MICROSEC = 1000;
public static final long NANOSEC = 1;
```

Figure 3.5: Timing constants defined in `org.jmlspecs.lang.DurationConstants`

```
/** This method creates a Vector2d that represents a voxel. */
/*@  public behavior
  @     requires position.x >= 0.0f && position.y >= 0.0f;
  @     duration 3 * MILLISEC;
  @ also
  @   public behavior
  @     requires position.x < 0.0f ^ position.y < 0.0f;
  @     duration 4 * MILLISEC;
  @ also
  @   public behavior
  @     requires position.x < 0.0f && position.y < 0.0f;
  @     duration 5 * MILLISEC;
  @*/
protected void voxelHash(Vector3d position,
                         Vector2d voxel) {
  int x_div = (int) (position.x / voxel_size);
  voxel.x = voxel_size * (x_div);
  if (position.x < 0.0f) voxel.x -= voxel_size;
  int y_div = (int) (position.y / voxel_size);
  voxel.y = voxel_size * (y_div);
  if (position.y < 0.0f) voxel.y -= voxel_size;
}
```

Figure 3.6: SafeJML method specification example. The method's specification has 3 specification cases, which are separated by `also`.

of the corresponding specification cases must have all their clauses satisfied by the method's execution (this is the reason the keyword is "also") [16]. For the duration clause, this allows the specification of a global worst case execution time (in a specification case with no precondition or precondition "`true`"), and more stringent constraints that are governed by other preconditions.

Figure 3.6 shows an example for the usage of the `duration` clause in multiple behavior specification cases for a method. These specification cases, cover three different scenarios for `x` and `y` in the position object, as distinguished by their requires clauses. In this example each specification case contains a requires clause and a duration clause. The duration clause will be used by the tool that is used to perform runtime checking of timing constraints for the method. This tool can check that the execution time for this method is no larger than the specified time for each case, given that the method's actual arguments satisfy the method's precondition. The tool also uses these expressions for timing analysis when this method is called. If at a particular call site it can be shown that the precondition of one specification case holds, then the corresponding duration clause can be used as the black-box value for the execution time of that call. (If at a call site one can only prove that one of several

```
1  /*@ splits_wcet true; @*/
2  void splitter(int arg1) {
3     /* ... */
4  }
5
6  /*@ splits_wcet arg1==3; @*/
7  void splitIf3(int arg1) {
8     /* ... */
9  }
10
11 void method1() {
12    splitter(1); // context-dependent analysis
13    splitter(2); // context-dependent analysis
14    splitter(0); // context-dependent analysis
15    splitIf3(1); // unified analysis
16    splitIf3(2); // unified analysis
17    splitIf3(3); // context-dependent analysis
18 }
```

Figure 3.7: SafeJML example using the `splits_wcet` clause.

preconditions hold, but not which one, then one must use the maximum of all the duration clauses in specification cases whose preconditions holds during checking.)

Context-Dependent Annotations Context dependency can aid the precision of WCET analysis if different calls of a method will have greatly differing timing behavior. SafeJML allows `splits_wcet` annotations on method specifications to specify when a certain tool should track the context of method calls. The syntax for the `splits_wcet` clause is as follows:

splits-wcet-clause ::= `splits_wcet` *predicate* `;`

The *predicate* in a `splits_wcet` clause can refer to any visible[6] fields and to the parameters of the method. The *predicate* clause instructs the tool whether a context-dependent static analysis should be used. If the predicate value is true, then the analysis tool must perform a context-dependent analysis at the place where the method was called.

Figure 3.7 shows how this clause is used in a method specification. In this example, calls to `splitter` will be considered a separate analysis path each time this method is called with a different set of arguments. Calls to method `splitIf3` however, are only considered for

[6]More information about visibility can be found in Section 2.4 of the JML Reference Manual.

jml-annotation-statement ::= ... | *loop-max-iter-stmt*
| *local-worst-case-stmt*
| *path-anchor-stmt*

Figure 3.8: SafeJML Annotation Statements.

a separate path analysis when the actual argument is 3, since that is the condition specified in its `splits_wcet` clause.

If the method `splitter` has distinct execution paths that depend on the actual arguments, then this clause will provide a tool with enough information to produce tighter WCET times by treating each execution path that depends on a specific parameter as if it is a separate method call. In other words, the `splits_wcet` clause doesn't ever cause violations to be recognized as errors. Instead, it provides a hint to the implementation so that it can produce a tighter WCET analysis.

3.2.3.2 Statement Annotations

SafeJML, like JML [17], allows several annotations to be written where statements may occur in Java code. Many of SafeJML's annotations are intended to be directly translated into inputs for various WCET tools. Three new statement annotations are identified in this section. These statements are added to the syntax of the *jml-annotation-statement* as shown in Figure 3.8.

Loop Maximum Iteration Statements In SafeJML a loop statement body can include a new annotation statement that specifies the maximum number of iterations of a loop. The syntax of the SafeJML annotated loop statements `maximum_loop_iterations` is as follows:

loop-max-iter-stmt ::= `maximum_loop_iterations` *constant-expression* `;`

This annotation statement is used to specify the maximum number of iterations the loop can have. This annotation has one argument, a *constant-expression*, which is a pure (side-effect free) Java expression that can be resolved at compile time, and is of type `long`. The expression's value specifies the maximum number of times that the body of the loop executes on any invocation, and one less than the number of times the loop test executes.

The example in Figure 3.9 includes a `maximum_loop_iterations` statement. In this example, the constant `MAX_ARR_SIZE` is used with prior knowledge from the programmer to bound the loop with an upper limit.

Local Worst Case Statements SafeJML has a new annotation construct for blocks of code nested inside conditional statements, the `local_worst_case` statement. This statement specifies the maximum number of times that a conditionally-guarded block of code can be executed, in total, during all iterations of the smallest enclosing loop (per activation of that loop). Such a statement must occur nested within a loop and before a conditional statement (such as an `if` or `switch` statement). The syntax of the `local_worst_case` statement is as follows:

local-worst-case-stmt ::= `local_worst_case` *constant-expression* `;`

The `local_worst_case` statement has one argument, a *constant-expression*, of type `long`. This argument denotes the maximum number of times that all paths on which the `local_worst_case` statement appears may execute during a single run of the enclosing loop. Any greater number of iterations is a violation of the specification. An example of the Local Worst Case Statement is shown in Figure 3.10 on the next page.

In this example, the body of the outer if-statement may execute at most 50 times (during each activation of the enclosing loop), and the inner if-statement may execute at most one time during the entire loop execution.

```
1  //@ public abstract model int MAX_ARR_SIZE = 100;
2
3  public abstract class SumArrayLoop {
4    public static long sumArray(int [] a) {
5      long sum = 0;
6      int i = a.length;
7      /*@ maximum_loop_iterations MAX_ARR_SIZE; @*/
8      while (--i >= 0){
9        sum += a[i];
10     }
11     return sum;
12   }
13 }
```

Figure 3.9: SafeJML maximum loop iterations statement annotation.

```
count = 0;
//@ maximum_loop_iterations 100;
for(i=0; i < limit; i++)
{
  //@ local_worst_case 50;
  if(buffer[i] == '*')
  {
    count++;
    //@ local_worst_case 1;
    if(count >= 50)
    {
      /* ... */
    }
  }
}
```

Figure 3.10: SafeJML local worst case statement annotations.

Path Annotations SafeJML supports the concept of specification of paths[7], and in particular which paths exclude which other paths.

Path annotations are based on the concepts of path names and path groups. A *path name* is an identifier that is declared to be associated with each path that includes the enclosing block. A *path group* is a collection of path names. Path groups are used to specify sets of execution paths.

Figure 3.11 shows the syntax for the *path-anchor-stmt*, which is used to declare path names and path groups. The keyword `path` is used to declare a path in the annotated code, and *path-name* identifier gives a name for this path. The keywords `\in` and `\exclude` are optional keywords that are used to include or exclude the declared path from a *path group*.

[7] A path is a possible execution sequence in a program.

path-anchor-stmt ::= `path` *path-name* [`\in` *path-group-name*]
[`\exclude` *path-group-name*] `;`
path-name ::= *ident* | `\default`
path-group-name ::= *path-name*

Figure 3.11: Syntax for Path Annotations.

To declare a new path group, a new name has to appear in the `\in` clause of the *path-anchor-stmt* statement. If the `\in` clause is not used, then the path group for that path is just the singleton group, whose name is the same as the path's path-name.

The main purpose for introducing path groups is to be able to optimize a WCET analysis by reducing the number of paths to be analyzed, and improve its precision by explicitly describing infeasible paths.

The path name `\default`, which is always in scope, is used to represent the default execution path of the code. Excluding a path from the `\default` path is useful when that path conforms to a special mode of operation that will never be used (in the system being built or as it will be deployed), or when the path should not be considered as part of the normal behavior of the system.

Figure 3.12 on the following page shows the usage of path names and path groups. Note that in line 4, we give the path inside the `if` statement a name, `char_found`, to indicate that this path will be executed only if the search method has a hit on the searched character. Similarly, in line 9, we specify a new path, `char_found2`, and we specify that this path is part of the implicit path group generated earlier, `char_found`. This technique urges the verification tool to consider those two paths as one, hopefully reducing analysis time. Thus, the `path` statement, just like `splits_wcet` statement, will not cause any error to be raised during the analysis, instead, it provides a hint to the implementation so that it can produce a tighter WCET analysis.

In line 13, we annotate a new path, and we exclude it from the original group that was implicitly declared earlier. This means that this path, `char_not_found` cannot be executed along with the previous two paths declared as part of the group `char_found`. This reduces the number of feasible paths to consider from 8 to 2. Thus less time and memory should be required to perform the analysis.

The last part of the example is line 17, which indicates that this part of the code is not reachable (at least in normal operation), hence, should be excluded from the analysis[8].

3.2.3.3 Duration Annotations for Blocks

As in JML, duration clauses, like other method specification contract clauses, can be applied to a block of code using the `refining` statement. The syntax for the `refining` statement [17, 81] is shown in Figure 3.13 on the next page.

[8] JML has a statement `unreachable;` which is an annotation that asserts that the control flow will never reach that point in the program[17, Section 13.4.4]. The semantics of `\exclude \default` are similar, as the later is just a hint for the analysis.

```
1  void search( int limit ) {
2     /* ... */
3     if(count != 0){
4        //@ path char_found \exclude char_not_found;
5        /* ... */
6     }
7     /* ... */
8     if (count !=0 ){
9        /*@ path char_found2 \in char_found; @*/
10       /* ... */
11    }
12    if (count == 0){
13       /*@ path char_not_found \exclude char_found; @*/
14       /* ... */
15    }
16    if (limit == 101){
17       /*@ path path_not_reachable \exclude \default; @*/
18       /* ... */
19    }
20 }
```

Figure 3.12: SafeJML path annotations used with if statements.

refining-statement ::= `refining` *spec-statement statement*
    | `refining` *generic-spec-statement-case statement*
generic-spec-statement-case ::= ... | *simple-spec-statement-body*
simple-spec-statement-body ::=
        simple-spec-statement-clause simple-spec-statement-clause

Figure 3.13: Syntax of Refining Statement From JML Reference Manual.

```
1     //@ refining
2     //@   duration 3 * MILLISEC;
3     { m(); }
```

Figure 3.14: The Usage of the Refining Statement.

The meaning of a statement `refining` S C is that the code C has to satisfy the specification S. For example, the code shown in Figure 3.14 says that the call to `m` may take at most 3 milliseconds to execute.

The use of such refining statements can help both designers and verification tools. Designers can use refining statements with duration clauses to allocate a method's time budget to individual blocks of code. Verification tools can also use refining statements that specify durations to better pinpoint timing errors. Verification tools can also use refining statements as context-sensitive specifications of the time that particular blocks of code may take. This can be useful when calling methods that do not have SafeJML specifications.

3.2.3.4 Error reporting

For statement annotations like the *loop-max-iter-stmt* and the *local-worst-case-stmt*, the JAJML compiler for SafeJML reports the violation to the user by throwing an error (a subtype of `JMLAssertionError`) as soon as such a violation is detected. However, for other constructs (like the *duration-clause*) a timing violation can only be flagged at program termination because their semantics under SafeJML allows them to be evaluated off-line by a tool after the program is run.

3.3 Design of the SafeJML Runtime Assertion Checker

To complete the system design, this section discusses the implementation of the runtime assertion checker.[9] It is not intended to provide a full implementation of the runtime assertion checker for SafeJML features in this chapter. Instead, Section 3.3.1 on the next page shows the implementation of the duration runtime checker.

[9] A wiki page for SafeJML is provided at `http://tinyurl.com/28zllux`. The page contains documentation on how to build and test SafeJML.

```
1  TransformMethod( md:MethodDecl, mb:MethodBody )
2    lst := new code block
3    if methodID of md exists
4      methodID := md.getMethodID()
5    else
6      methodID := md.createMethodID()
7    specCases := new List()
8    durationCheckConditions := new List()
9    for each specCase in md.getTypeHierarchy.getSpecCases()
10     durationMin := specCase.durClause.getMinExpression()
11     durationMax := specCase.durClause.getMaxExpression()
12     requiresClause := specCase.getPrecondition()
13     durationCheckConditions.add(requiresClause, durationMin,
          durationMax)
14   durationCheckerMethod := createMethod(durationCheckConditions)
15   md.hostType.addMethod(durationCheckerMethod)
16   startTimer := new statement ("CheckDurationData", "enter",
        methodID, "NanoClock.getCurrentTime()")
17   endTimer := new statement ("CheckDurationData", "exit", methodID,
        "NanoClock.getCurrentTime()")
18   newMethodBody := createMethodWrapper(md, mb)
19   lst.add(createMethodCall(durationCheckerMethod))
20   lst.add(startTimer)
21   lst.add(newMethodBody)
22   lst.add(endTimer)
23   md.setMethodBody(lst)
```

Figure 3.15: Transformation Algorithm for Annotated Methods.

3.3.1 The Design of the Duration Checker

Refer to the overall system architecture shown in Figure 3.1 on page 22. As discussed earlier in Section 3.1, The analysis process consists of four stages. During the first stage of the analysis process, the SafeJML compiler invokes transformation algorithms on the Abstract Syntax Tree (AST) of the source code. This results in code transformations to the methods annotated with the duration clause. The algorithm used for annotated methods transformation is shown in Figure 3.15.

This algorithm transforms the duration clause for the method of interest or that is defined in a supertype with an implementation of that method. Cheon and Leavens [82] discussed the implementation of a runtime assertion checker for JML and its complications. They described two major problems associated with implementing specifications inheritance. The first problem is the inability of a subtype's assertion checking method to determine the

existence of a corresponding specification in its supertypes. This is due to the fact that supertypes can be precompiled. To solve this problem, we require that all duration clauses in supertypes must be compiled using the SafeJML compiler during analysis.

The second problem is the potential for infinite loops if assertions are checked during assertion checking, as noted by Meyer [53]. Therefore, if a method is called during assertion checking of another method, and the called method has its own specification, then these checks are not performed, in order to avoid potential infinite checking loops where checking the second method's specification might call the first method. Similarly, duration analysis will not be performed by SafeJML on the methods called during duration checking of another method.

During compilation, the method of interest is transformed to perform assertion checks for all method specification clauses. The algorithm for transformation of a method's code is shown in Figure 3.15 on the preceding page. This algorithm is based on Cheon's approach [82]. It was first implemented in the original implementation of the JML RAC, and was reimplemented in the JAJML RAC tool to implement method specification checks. The algorithm is altered to include duration checking analysis for the SafeJML RAC. The algorithm starts by checking if a method ID was created for that method as a model field; if this is true, then that method ID is retrieved and used to mark the duration records that will be saved later. If no method ID was found, then the algorithm creates a new method ID in a model field. Then the algorithm desugars all specification cases in the method of interest and all other specification cases in its type hierarchy. These specifications are used to generate assertion check methods for pre and post conditions as well as duration clauses. The algorithm then injects entry and exit method calls before and after a wrapper method that contains the original method body. The declarations for these methods are shown in Figure 3.16.

Figure 3.17 on the following page shows how the example from Figure 3.6 on page 28 is transformed using the SafeJML RAC compiler. Figure 3.18 on page 39 shows the duration checker method that allows the duration checker to decide whether a duration clause must be checked based on the precondition of a specific specification case.

```
void CheckDurationData.enter(int mID, long ts);
void CheckDurationData.exit(int mID, long ts);
```

Figure 3.16: Declaration for `enter` and `exit` Methods Injected by the SafeJML Compiler for SafeJML Annotated Methods.

```
1     //@ requires position.x >= 0.0F && position.y >= 0.0F;
2     //@ duration 0 .. 3000000;public
3     //@ requires position.x < 0.0F ^ position.y < 0.0F;
4     //@ duration 0 .. 4000000;public
5     //@ requires position.x < 0.0F && position.y < 0.0F;
6     //@ duration 0 .. 5000000;public
7   protected void voxelHash(Vector3d position, Vector2d voxel) {
8     checkPre$voxelHash$Reducer(position, voxel);
9     try {
10      updateDurLimits$voxelHash$Reducer();
11      this.chkdur$.addEntryRecord(cdx.Reducer.m$1, NanoClock.now());
12      internal$voxelHash(position, voxel);
13      this.chkdur$.addExitRecord(cdx.Reducer.m$1, NanoClock.now());
14    }
15    catch (org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError
         rac$e) {
16      throw new
           org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionError(rac$e);
17    }
18    catch
         (org.jmlspecs.jmlrac.runtime.JMLExitNormalPostconditionError
         rac$e) {
19      throw new org.jmlspecs.jmlrac.runtime.
20            JMLInternalNormalPostconditionError(rac$e);
21    }
22    catch
         (org.jmlspecs.jmlrac.runtime.JMLExitExceptionalPostconditionError
         rac$e) {
23      throw new org.jmlspecs.jmlrac.runtime.
24            JMLInternalExceptionalPostconditionError(rac$e);
25    }
26    catch (org.jmlspecs.jmlrac.runtime.JMLAssertionError rac$e) {
27      throw rac$e;
28    }
29  }
```

Figure 3.17: The Method in Figure 3.6 on page 28 Transformed by the SafeJML RAC.

```
1   private transient synthetic boolean rac$pre$voxelHash$0;
2   private transient synthetic boolean rac$pre$voxelHash$1;
3   private transient synthetic boolean rac$pre$voxelHash$2;
4   private transient synthetic CheckDurationData chkdur$ =
       CheckDurationData.getInstance();
5   private static final transient synthetic int m$1 = 1;
6   synthetic private void updateDurLimits$voxelHash$Reducer() {
7     this.chkdur$.EnterNewRecord(cdx.Reducer.m$1);
8     if(this.rac$pre$voxelHash$0)
9       this.chkdur$.populateUpperLowerLimit(cdx.Reducer.m$1, 0, 0,
           3000000);
10    if(this.rac$pre$voxelHash$1)
11      this.chkdur$.populateUpperLowerLimit(cdx.Reducer.m$1, 1, 0,
           4000000);
12    if(this.rac$pre$voxelHash$2)
13      this.chkdur$.populateUpperLowerLimit(cdx.Reducer.m$1, 2, 0,
           5000000);
14  }
```

Figure 3.18: The duration checker method generated for the transformed method in Figure 3.17 on the preceding page.

After transformation, and during execution, the trace methods produce a record which contains timing information about the method entry and exit. This record consists of the following fields:

- `mID`: Represents the method ID. This value is injected by the compiler using a model field.
- `ts`: This is the time at which the entry and exit methods are executed, and thus represents the entry and exit timestamps of the method.
- `durMin`: This value is only recorded in the exit record, it holds the value of the minimum specified value in the duration clause.
- `durMax`: This value is only recorded in the exit record, it holds the value of the maximum specified value in the duration clause.

Several implementations of these methods for collecting traces are supported by the SafeJML tool. For example, if the code is going to be simulated on a PC, the programmer then will choose to collect traces in the PC's memory, and then dump these traces in files to be analyzed later. Similarly, if the programmer is running the trace collection algorithm on the host hardware, the traces can be directed into a serial port and then collected on different

hardware. The net result will be a trace file to be used in the next step of the process. Both of these methods are supported by the implementation. Furthermore, a collection of "static native" trace handlers are available in FijiVM, which gives one the freedom to change the trace collection implementation as desired.

This approach has a problem when a method does not terminate normally. For example, if an exception is raised during a method execution, then there is no guarantee that the trace will be generated. This will cause the analysis to fail or produce inaccurate results. I presently sidestep this issue and effectively require that all methods (on which the tool performs duration analysis) terminate normally. In particular, this means that every exception must be caught and handled within the method itself. In the future I plan to overcome the problem of exceptional termination by using Java's try-finally statements to capture the exit trace of the method. However, due to an incompatibility issue with the byte code produced by SafeJML and the Fiji compiler, I could not currently implement this approach. Another approach is to start a timer in a separate thread to protect against infinite loops, and have that thread throw an exception when the method takes longer than specified. The evaluation and implementation of either of the aforementioned approaches is left for future work.

During the third stage of the analysis, the code is executed. This will cause the analysis to be performed and will result in producing execution traces. As some real-time systems are designed to run as infinite loops, there has to be a termination point for the purposes of the analysis. At the end of the execution, the analysis flushes the trace information into a trace file. This file will be used later to execute the checking part of the algorithm.

At the fourth stage of the analysis, the execution traces are processed by the SafeJML checker. In addition to the duration information, execution traces contain information about preconditions. This stage analyzes the trace files and compares them to specifications. The output of this stage is a report of all violations of the duration specifications. The SafeJML checker stage checks execution traces against the specifications in the program by calculating differences in timestamps between the entry and exit traces of each method call. This happens only if the precondition at that specific instance of execution is true.

3.3.2 Discussion

The approach described in this section to develop a runtime assertion checker for duration annotations introduces a practical problem. This problem is realized by the extra processing required during execution time to be able to collect all needed information for the checker to be able to do its job correctly. This is the case as the duration expression calculation requires information that is available only at runtime. Clearly, evaluating these methods at execution time is a process that takes time, and will affect the overall timing

analysis of the program. This is true if two nested method calls are being analyzed simultaneously. A practical and precise solution remains an open topic for research and future work.

3.4 Related Work

Many researchers have worked on the problem of WCET analysis. One early work in this area is Shaw's book [83]. Shaw introduces a method for precise analysis using path expressions to perform measurements. Shaw does not consider subtyping. Furthermore, this analysis is not modular, since it does not use specifications.

Schoeberl and Pedersen [84] describe a precise WCET static analysis for Java systems based on the Java Optimized Processor (JOP). Like Shaw's analysis, it is a whole program static analysis; however Schoberl's analysis is path insensitive. Also like Shaw's analysis, the tool uses integer linear programming to find WCET solutions from Java bytecode. Java bytecodes for the JOP are predictable due to JOP's design, which supports predictable timing behavior by design. Since it is a whole program analyzer, Schoberl's tool is not modular. Furthermore, it only handles subtype polymorphism by taking the worst case over all possible method calls. SafeJML allows the use of supertype abstraction to handle subtype polymorphism, as will be shown in the next chapter.

Formal verification of timing specifications is also introduced in Hehner's work [85]. Hehner used refinement calculus to formalize the verification of timing specifications, but he does not discuss OO issues such as subtyping.

The design and implementation of the `duration` clause in SafeJML (and in JML) is based on the work of Krone *et al.* [19, 80].

Much of SafeJML's design is built to accommodate the RapiTime tool [86, 76], a hybrid analysis tool used to perform hybrid WCET analysis for C programs. The heavy influence of RapiTime on SafeJML largely results from our desire to translate SafeJML into RapiTime's input language. However, RapiTime does not in itself deal with SCJ, nor does it have facilities for specification of functional behavior. SafeJML, since it builds on JML, has extensive facilities for specification of functional behavior (which we have largely ignored in this chapter). Such specification facilities may be quite useful for safety-critical systems.

Gustafsson *et al.* [87] suggest using abstract interpretation to aid WCET analysis. They introduced their idea first in their previous work [88]. They introduce a tool called SWEET, which uses abstract execution to automatically derive loop bounds and infeasible paths from C programs. SWEET is integrated with a compiler and performs its analysis on the intermediate representation of the compiler, which makes its usage limited to code compiled using that compiler. Such automatic derivation of annotations reduces manual intervention and thus makes the analysis process easier, less error prone and more accurate.

SafeJML could benefit from abstract execution to reduce the need for many of the annotations that are designed to help RapiTime. However, SWEET itself does not treat SCJ programs and since it is a whole-program analysis, is not modular.

Hu *et al.* [89] present a static timing analysis approach for hard real-time systems based on RTSJ. They introduced XRTJ (Extended Real-Time Java), an architecture based on RTSJ to implement static timing analysis in a portable code context. Their work can be merged with other approaches such as model checking. Their approach uses annotations similar to JML annotations to inject properties that can be checked at development time. They also implement WCET analysis based on these annotations. However, this work is tightly integrated with RTSJ and therefore does not apply to SCJ programs.

In the Extended Static Checker for Java (ESC/Java), Flanagan *et al.* [69] use the JML `assume` statement to warn about infeasible paths. Essentially, the statement annotation `//@ assume false;` mean that the current path is infeasible, and the tool will give a warning if this statement can be reached. However, JML and the ESC/Java tool do not provide full support for path names and path groups as addressed in Section 3.2.3.2 on page 32. Features like the one provided in SafeJML provide full control over path analysis and therefore is better for users.

CHAPTER 4
SUBTYPING AND WCET ANALYSIS

In this chapter[1], a novel approach to use SafeJML to specify subtypes in SCJ programs is introduced. This approach solves the problem of modular subtype specifications for SCJ programs.

Recall that SCJ allows for subtype polymorphism, also known as dynamic dispatch, which determines the code to run for a method call such as `o.m(x)` based on the dynamic class of the receiver object, `o`. But due to the problems with reasoning about subtype polymorphism that was described in Section 1.1.2, some researchers in real-time systems suggest that this feature should be disallowed [25]. However, in this section, a solution to these reasoning problems is introduced by using standard techniques for modular reasoning in the presence of subtype polymorphism, and how these techniques can be applied to reasoning about timing constraints in SCJ programs. Allowing the use of subtype polymorphism should also have benefits in terms of programmer efficiency and ease of maintenance, since it allows code reuse [90] and the use of object-oriented design patterns.

A standard methodology for modular reasoning in the presence of subtype polymorphism, called *supertype abstraction* [16, 28], is to use the static types of each call's receiver to find the specifications for reasoning about the effect of a method call. For method calls, supertype abstraction says that to verify $\{P\}$`o.m();`$\{Q\}$ (that starting in a state that satisfies property P, the call `o.m()` necessarily achieves a state that satisfies Q) one uses the specification of `m` associated with the static type of `o`. (In particular, one checks that P implies the precondition specified for `m` in the static type of `o`, and that this method's postcondition implies Q.)

Soundness of supertype abstraction requires types to be behavioral subtypes of their supertypes [27]. If S is a subtype of T, then S is a *behavioral subtype* of T only if all overriding methods in S obey their specification in T [91, 92, 16, 26, 27, 28, 93].

However, applying behavioral subtyping to timing constraints poses a practical problem for timing constraints. This problem arises because methods in a subtype are often required to do more elaborate information processing than the methods they override in their supertype(s). This often occurs because a subtype's instances may contain more information than those of its supertypes. For example, consider again the type `Vector2d` and its subtype `Vector3d`. Since the `scale` method of `Vector2d` has a very tight timing constraint,

[1]This chapter is based on our paper "Specifying Subtypes in SCJ Programs" [24].

which permits just enough time for a reasonable implementation, that specification does not allow enough time to perform the calculations needed in a `Vector3d` object.[2]

As explained in Section 1.1.2, one can try to avoid such problems by either not overriding methods, or by underspecification. Those ways of avoiding the problem are orthogonal to SafeJML; that is, SafeJML's analysis will still work correctly if these techniques are used. However, because these approaches either give up on method overriding or use imprecise specifications, they either give up some of the flexibility of OO programming or result in an imprecise analysis.

4.1 A Solution for Subtyping

The key to solving this problem is to recognize that the problem lies in seeking an *a priori* fixed limit on the method's time bound. That is, timing constraints for methods cannot simply be constants. This is not a new observation, as others have already noted that timing constraints in general must depend on data such as arguments to a method [94, 19, 80]. As the examples shown previously illustrate, the runtime type (i.e., class) of a method's receiver object is also data that is input to that method. Thus the timing constraint of a method in general may need to depend on the receiver's dynamic type.

For runtime checking tools like SafeJML, using this technique requires collecting information about timing constraints at two stages: first during the compilation of the timing constraints, and second during execution. SafeJML checks timing constraints after the program has finished running, when it analyzes the collected information and compares the observed execution times to the specified timing constraints.

The insight that the receiver's dynamic type is input to each method dovetails with an elegant specification technique first published by Matthew Parkinson [95, 96]. The essence of Parkinson's technique is to write specifications of methods using "abstract predicate families" [95, p. 78], which can have differing definitions in various types. The value of an abstract predicate can (and generally does) depend on the runtime type of a method's receiver. The SafeJML analog of an abstract predicate family is a non-static `pure model` method; such a model method can only be used in specifications, and yet can be overridden in different types, which allows it to have a different meaning for each subtype. To interpret such model methods during SafeJML's checking (which takes place after running the program), SafeJML must record information about the runtime type of each method's receiver; this information allows SafeJML to use the correct model method implementation to interpret a specification.

[2] If one thinks about subtypes that are for higher dimensional vector spaces in general, then one realizes the truly fundamental nature of this specification problem.

Consider the example introduced earlier in Figures 1.3 and 1.4 in Section 1.1.2, the timing constraint for `Vector2d`'s `scale` method can be rewritten in JML by introducing a specification like the following, where `scaleTime` is a `pure model` method.

```
//@ duration scaleTime();
```

The meaning of the method `scaleTime` would change, using overriding, in each concrete subtype. To reason about the time taken by calls to `scale` either requires knowledge of the exact runtime type of the receiver, or some separate specification of how the `scaleTime` method depends on the dynamic type of the receiver. This dependency can be captured in the specification of the `scaleTime` method for a specific type, which would then apply to all its subtypes. For example, if one uses Figure 4.1 as the specification for `scale` and additionally specifies that the result of `scaleTime` is no greater than the number of dimensions in that vector object times the time it takes for the platform to compute one floating point multiplication and one floating point assignment, then this specification would have to be obeyed by all subtypes of the class `Vector2d`. Thus, one can reason using static type information (supertype abstraction [28, 16, 27]). However, if one needs more precision, and if during reasoning one can prove something about the exact dynamic type of a collection, then one can instead use the specification for a type that is an upper bound on the object's exact dynamic type and that type's `scaleTime` and `getDimensions` methods.

How the proposed solution applies to the vector example is shown in Figures 4.1 and 4.2. The `pure model` methods `scaleTime` and `getDimensions` are introduced in `Vector2d` and used in the duration specification rewritten for `Vector2d` and inherited by `Vector3d`, which overrides `getDimensions`.

For runtime checking of timing constraints, it is important that the implementation consumes minimal (and constant) time. The implementation achieves this by outputting an execution trace, during the program's execution, with enough timing information to enable later check of timing constraints. A similar technique was used to evaluate the performance of oSCJ implementation [78], and Alan Shaw described a similar technique in his book [83]. When the program is not executing, the tool checks the program's execution trace and compares the duration of method calls with those specified for the corresponding method.

The novel feature of this implementation is that execution traces are designed to contain enough information about the program's state to enable checking duration clauses that use abstract predicates (model methods), which depend on the dynamic type of the receiver.

For this to happen, the algorithm introduced in Section 3.3.1 on page 36 is extended to collect information about methods overridden is subtypes and are included in the duration analysis. This is done by extending the trace record introduced earlier to support collecting information about such methods. This is realized by adding a new field to the record, `classDynamicType`, which represents the receiver's dynamic type. This value is tracked by a variable injected by the compiler.

```
1  public class Vector2d {
2      //@ public model JMLDataGroup dims;
3      protected /*@ spec_public @*/ float x, y; //@ in dims;
4
5    /*@   public normal_behavior
6      @     requires !Float.isNaN(factor);
7      @     assignable dims;
8      @     ensures x == \old(x) * factor && y == \old(y) * factor;
9      @     duration scaleTime();
10     @*/
11     public void scale(float factor) {
12         this.x *= factor;
13         this.y *= factor;
14     }
15
16   /*@
17     public pure model long scaleTime() {
18         return this.getDimensions()
19              * (ITimeConstants.MultiplyTime +
                    ITimeConstants.AssignTime);
20         }
21     @*/
22
23   /*@ ensures \result >= 2;
24     public pure model int getDimensions() {
25         return 2;
26     }
27     @*/
28 }
```

Figure 4.1: Specifications for `Vector2d`, modified from those in Figure 1.3 to use the proposed approach with model methods (following Parkinson *et al.*).

```
1  public class Vector3d extends Vector2d {
2
3      protected /*@ spec_public @*/ float z; //@ in dims;
4
5    /*@ also
6      @   public normal_behavior
7      @     requires !Float.isNaN(factor);
8      @     assignable dims;
9      @     ensures z == \old(z) * factor;
10     @*/
11     public void scale(float factor) {
12         super.scale(factor);
13         this.z *= factor;
14     }
15
16     /*@
17     public pure model int getDimensions() {
18         return 3;
19     }
20     @*/
21 }
```

Figure 4.2: Specifications for `Vector3d`, modified from those in Figure 1.4 to use the proposed approach.

This extension makes the compiler aware of the overridden methods in subtypes. This enables the checker to check the duration clause depending on the dynamic type of the receiver.

4.2 Analyzing Subtypes in SCJ using SafeJML

As introduced in Section 2.4, SCJ specifications introduce the concept of release parameters that are meant to provide an estimate of the WCET time needed to execute a schedulable object. In this section, the relationship between SafeJML specifications and the SCJ release parameters mechanism is introduced using an example.[3] This example is inspired by one of the SCJ examples that ships with the FijiVM implementation. The connection to the SCJ release parameter mechanism can be seen in the mission class, shown in Figure 4.3. The mission class `ListHandlerMission` initiates two list handler objects: one of type `ListHandler` (shown in Figure 4.4 on the following page) and the other of type `SortedListHandler` (shown in Figure 4.5 on page 50). Both of these handler types are subtypes of `PeriodicEventHandler`.

```
1  @SCJAllowed(members=true, value=LEVEL_2)
2  public class ListHandlerMission extends Mission {
3     private static final int MISSION_MEMORY_SIZE = 10000;
4     public void initialize() {
5        ManagedMemory.getCurrentManagedMemory().resize(MISSION_MEMORY_SIZE);
6        (new ListHandler("lsthndlr",
7               new RelativeTime(0, 0),
8               new RelativeTime(0.5 * ITimeConstants.ObjectSwapTime,
                   0)))
9               .register();
10       (new SortedListHandler("srtlsthndlr",
11              new RelativeTime(0, 0),
12              new RelativeTime(ITimeConstants.ObjectSwapTime
13                   * n * (Math.log(n), 0))).register();
14    }
15 }
```

Figure 4.3: Mission class that uses `ListHandler` and `SortedListHandler` objects.

SafeJML and SCJ's specification's are incompatible at the level of periodic event handlers. First, it is required for any type that extends `PeriodicEventHandler` to have its

[3]More information about SCJ release parameters can be found in Section 2.4 and detailed discussion can be found in the SCJ Specifications [46].

release parameters defined. Adding SafeJML specifications to such SCJ release parameter specifications would result in duplication of specification of timing constraints, which would potentially cause maintenance problems. Second, because of that potential duplication of specifications and checks, if one were to use SafeJML to specify timing constraints at the periodic event handler level, then any missed deadlines would be noted twice: once by the SCJ miss handler and once by the SafeJML checker. Thus, we recommend that SafeJML not be used to specify the `handleAsyncEvent` method, instead, SafeJML can be used to specify more detailed timing constraints.

The ability of SafeJML to specify timing constraints at more detailed levels, including individual methods, is illustrated by our specifications for the two types of lists in this example. Figure 4.6 on page 51 declares a type `UnsortedList` that inverts a list when the method `process()` is called. It is used by the handler type `ListHandler`.

In Figure 4.7 on page 52, we introduce another type, `SortedList`, which is a subtype of `UnsortedList`. This is used by the handler type `SortedListHandler`. When `SortedList`'s method `process()` is called, the `SortedList` object sorts the list instead of inverting it.

```
1  public class ListHandler extends PeriodicEventHandler {
2
3     static final int priority = 13, mSize = 10000;
4     protected /*@ spec_public @*/ UnsortedList list;
5     // ...
6
7     public ListHandler(String hdlName,
8           RelativeTime startTime,
9           RelativeTime period) {
10       super(new PriorityParameters(priority),
11          new PeriodicParameters(startTime, period),
12          new StorageParameters(mSize, mSize, mSize));
13       list = new UnsortedList();
14    }
15    // ...
16    public void handleAsyncEvent() {
17       // ...
18       list.process();
19       // ...
20    }
21    // ...
22 }
```

Figure 4.4: Timing constraint specifications for `ListHandler` using SCJ method.

```
1  public class SortedListHandler extends PeriodicEventHandler {
2
3     static final int priority = 13, mSize = 10000;
4     protected /*@ spec_public @*/ SortedList list;
5     // ...
6     public ListHandler(String hdlName,
7           RelativeTime startTime,
8           RelativeTime period) {
9        super(new PriorityParameters(priority),
10          new PeriodicParameters(startTime, period),
11          new StorageParameters(mSize, mSize, mSize));
12       list = new SortedList();
13    }
14    // ...
15    public void handleAsyncEvent() {
16       // ...
17       list.process();
18       // ...
19    }
20    // ...
21 }
```

Figure 4.5: Timing constraint specifications for `SortedListHandler` using SCJ method.

```
1  public class UnsortedList{
2     // ...
3     protected /*@ spec_public @*/ List list;
4     // ...
5
6     /*@ public normal_behavior
7       @   requires list != null;
8       @   ensures list.size() == \old(list.size());
9       @   duration  processTime();
10      @*/
11    public void process() {
12       // ...
13       list.invert();
14       // ...
15    }
16    /*@
17    public pure model long processTime() {
18        return (long) (0.5 * ITimeConstants.ObjectSwapTime);
19        }
20    @*/
21    // ...
22 }
```

Figure 4.6: Specifications for `UnsortedList`.

The specifications for both types in Figure 4.6 on the preceding page and Figure 4.7 show how SafeJML's methodology can be used to specify timing constraints. For the subtype `SortedList`, since it is a behavioral subtype of its supertype `UnsortedList`, specifications from both types must be satisfied. However, following our proposed approach, the duration clause is only specified in the supertype, and only the pure model method, `processTime`, used by the duration clause, is overridden in the subtype in order for the duration clause to be satisfied.

4.3 Related Work on Subtyping

We know of no other solutions to the problem of subtyping for timing specification and verification.

Our solution for the subtyping problem was inspired by the work of Parkinson and his coauthors [96, 95]. In his work, Parkinson introduced the concept of abstract predicate

```
1  public class SortedList extends UnsortedList {
2     //@ public model instance int n;
3     //@ public initially n == 0;
4     //@ public invariant n >= 0;
5     //@ protected represents n = list.size();
6
7    /*@ also
8      @  public normal_behavior
9      @    requires n >= 1;
10     @    ensures list.isSorted();
11     @*/
12    public void process() {
13       // ...
14       list.sort();
15       // ...
16    }
17    /*@
18    public pure model long processTime() {
19        return (long)(ITimeConstants.ObjectSwapTime * n *
             Math.log(n));
20    }
21    @*/
22    // ...
23 }
```

Figure 4.7: Specifications for `SortedList`, a subtype of `UnsortedList`.

families, which allow specifications to vary based on the runtime class of a method's receiver. The use of abstract predicate families allows subclasses to have what seem like strikingly different behaviors, and allows for "reuse subtypes" that exploit inheritance even though they have what may seem like somewhat different functional behavior. A major difference between SafeJML's model methods and abstract predicate families is that model methods cannot be used to enforce permissions and do not by themselves enforce data abstraction. In Parkinson's work, inherited methods must be reverified in each subtype, unless the abstract predicate families used in the specification of a method are constrained by a reflexive and transitive relation [96, Section 6.5] with certain properties. One kind of relation that works is when abstract predicate family definitions are unchanged in a subtype, and another is that subtype's definitions are proper extensions. Although Parkinson and his coauthors do not extend their work to timing specifications, it seems that the idea of proper extensions would not normally apply to timing constraints, since, as we have argued, subtype methods often need more time to do their work than the methods they override. It is unclear whether the formal framework of abstract predicate families would work with our examples. Indeed, Parkinson leaves the problem of static reasoning with abstract predicate families largely open.

Schoeberl and Pedersen [84] describe a precise WCET for Java Systems based on the Java Optimized Processor (JOP). This is also a whole-program static analysis, which makes it non-modular. This analysis hides the problem of subtyping because it depends on the duration for execution of byte codes, which requires prior knowledge about each type at compile time.

The SARTS [97] tool implements a model-based schedulability analysis of SCJ systems. The tool automatically translates SCJ code implemented based on JOP, into an abstract time-preserving UPPAAL model, and then performs WCET verification using the UPPAAL model checker. SARTS is a static analysis tool, which makes it precise, but at the cost of analysis time. On the other hand, in SARTS, the level of accuracy can be adjusted to limit state space and verification time. (The size of the state space that SARTS can handle is limited by the capacity of the UPPAAL model checker's implementation [97, Section 7].) SARTS is also limited to specific hardware, namely the FPGA implementation of JOP [98].

Hu *et al.* [99] addressed dynamic dispatching in object-oriented and hard real-time systems. Their work was based on the XTRJ architecture [100]. This work utilizes annotations first introduced in the XTRJ architecture to perform WCET analysis for subtypes and dynamically-dispatched methods. However, handling dynamic dispatching is performed by assuming that the caller has enough information about each call's receiver exact type.

CHAPTER 5
SYSTEM EVALUATION

This chapter presents the evaluation of SafeJML based on the four requirements identified in Chapter 1. These are expressiveness, precision, direct feedback, and developer efficiency. The programs used in this evaluation are based on examples from the FijiVM distribution.

5.1 Experiment Setup

The main example from FijiVM distribution that was used in this chapter is the `CDx` [20] benchmark. The CD (Collision Detector) benchmark suite is an open source application that implements an aircraft collision detection algorithm with simulated data. The benchmark comes in different flavors to support different hardware and software platforms. One flavor is a C-based implementation, which was originally developed as part of this dissertation to evaluate RapiTime[76]. For the purpose of this evaluation, a different flavor of the CDx was used; `minicdx` [78], which is provided as an example in the FijiVM distribution of the oSCJ implementation.

The minicdx example implements one periodic task; the collision detector task. This task runs on a pregenerated set of frames that describe a number of planes and their 3D coordinates in space. The task then works on detecting the number of collisions that could occur in a specific time frame[1].

The minicdx algorithm uses a simulator to generate workloads to be able to test the software. In this experiment, the simulator is set to generate 1000 frames, each has a preconfigured number of planes, and the simulation is performed at a constant interval of 50ms.

Figure 5.1 on the following page show the locations of six planes in the experiment. The figures to the left show the X and Y locations of the planes for a subset of the run (the first 300 frames). Location information is generated using predefined formulae to guarantee collision occurrence for testing purposes. In this case, a star formation of the planes movement guarantees collisions at different intervals.

[1] Algorithm details and test results are available in T. Kalibera [20] paper.

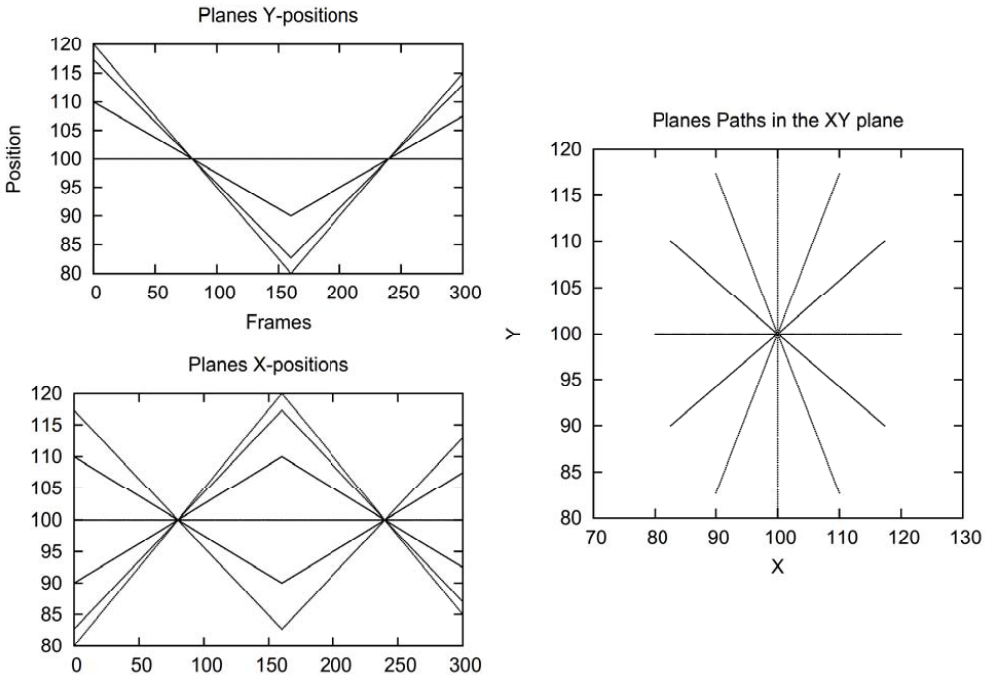

Figure 5.1: Locations for Six Planes used in the experiment

For the experiments presented in this chapter, a laptop with Intel Core 2 Duo x86 CPU running at 2.4GHz with 4GB of RAM was used. The machine has Ubuntu 11.10 32-bit Linux installed. A memory-based implementation was used to collect all data points used for the experiments. While this approach limits the number of data points that can be collected for analysis, it is considered a more accurate method for analysis if compared to other methods like disk-based or serial port collection methods. Plsek *et al.*[78] compared the performance of the benchmark used in this evaluation; the `minicdx` and the original benchmark, `CDx`, both written in Java. The comparison was performed on both LEON3 and x86 architectures. The comparison shows strong correlation of execution times between the two benchmarks in both architectures.

5.2 Evaluating Expressiveness

In order for the duration clause to be useful, the timing analysis resulting from the use of the duration clause must first be expressive. In this context, expressiveness means that SafeJML must be useful for expressing specifications of timing constraints for real safety-critical and real-time systems.

First step in evaluating expressiveness is to use the `duration` annotation to annotate the methods of interest in the project. Eight methods were selected to be annotated and analyzed. Figure 5.2 shows the call hierarchy for the methods being annotated.

```
1 TransientDetectorScopeEntry.run()
2   TransientDetectorScopeEntry.createMotions()
3     Aircraft.getCallsign()
4   TransientDetectorScopeEntry.lookForCollisions()
5     TransientDetectorScopeEntry.reduceCollisionSet()
6       Reducer.performVoxelHashing()
7     TransientDetectorScopeEntry.determineCollisions()
8       Motion.findIntersection()
```

Figure 5.2: Call hierarchy for the methods annotated in the miniCDx project.

As a start, the first method in the hierarchy, `TransientDetectorScopeEntry.run()`, was annotated. This is the main method that implements the CD task. The method is shown in Figure 5.3. This method is called from the object of type `TransientDetectorScopeEntry`, which is instantiated and called from an object of type `CollisionDetectorHandler`, a subtype

```
1   /*@   public normal_behavior
2     @     requires true;
3     @     assignable reducer,numberOfCollisions,
4     @                cdx.ImmortalEntry.detectedCollisions;
5     @     duration Constants.DETECTOR_PERIOD;
6     @*/
7
8   public void run() {
9     final Reducer reducer = new Reducer(voxelSize);
10    int numberOfCollisions = lookForCollisions(reducer,
        createMotions());
11    if (cdx.ImmortalEntry.recordedRuns <
        cdx.ImmortalEntry.maxDetectorRuns) {
12      cdx.ImmortalEntry.detectedCollisions
13        [cdx.ImmortalEntry.recordedRuns] = numberOfCollisions;
14    }
15  }
```

Figure 5.3: Specifications for the method `run()` in `TransientDetectorScopeEntry`, a class used to implement the collision detector algorithm in minicdx benchmark.

of `PeriodicEventHandler`. This is the standard way to establish a periodic activity, and thus permits the automatic execution of SCJ code[2].

To perform the experiment, I ran the SafeJML tool on the example and analyzed the resulting duration information. Figure 5.4 on the following page shows a histogram of the runtime of the afoementioned method in 10 experiments. The results show that all runs took less than 0.22ms to conclude. This result is far below the specified duration (50ms), hence, the tool did not report any error for the experiments. Results show that about 96% of the runs took less than 0.12ms. While this shows that the system did not have any misses, the results show that about 3.5% of the runs took between 0.14 and 0.16ms. These runs are the ones that had collisions, hence the extra time spent detecting these collisions. During the majority of the runs, the system was not able to detect any potential collisions, so the collision detection process terminates earlier and takes less time to execute than the specified time.

To complete the experiment, all the other methods listed in Figure 5.2 on the previous page were annotated with the `duration` clause. This experiment revealed two interesting observations in the implementation.

[2]Refer to Section 2.4 on page 16 about SCJ background for more information about the architecture of SCJ programs.

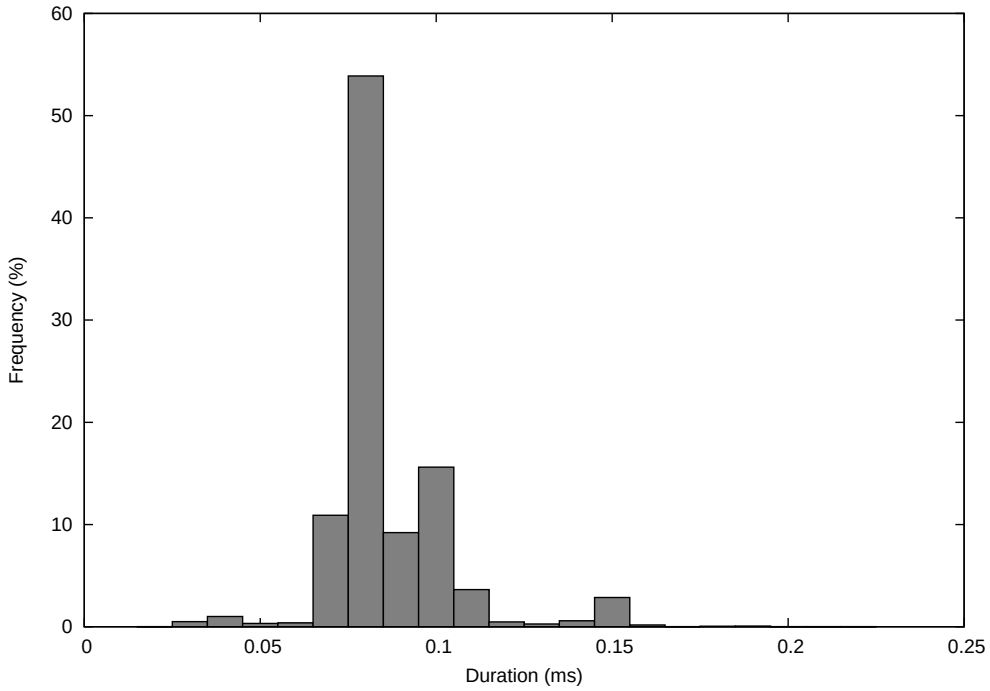

Figure 5.4: Frequency of durations of 20 experiments for the annotated method

- None of the annotated methods required specifying a minimum limit for the duration clause. This observation does not mean that this feature is not useful. Other systems specifications call for this feature explicitly[3]. However, the miniCDx system did not require this feature to be used. To address this issue, the analysis in Section 5.3.3 on page 66 introduces a hypothetical minumim limit to the methods shown in the analysis in order to prove the ability of the tool to analyze minimum limits.

- An attempt to run the analysis on a caller and a called method that are both annotated using the `duration` clause caused the analysis to report a false missed deadline for the caller method. This behavior is related to the algorithm used in the system implementation. This is handled currently by warning the user that nested analysis is taking place.

To further analyze the second observation, I ran the analysis on a three-level nested call hierarchy of the methods shown in Figure 5.2 on page 56. Three methods were annotated, `TransientDetectorScopeEntry.run()`, `TransientDetectorScopeEntry.lookForCollisions()` and `TransientDetectorScopeEntry.reduceCollisionSet()`. The experiment was run three times, once by annotating the three methods, second by annotating the first and second methods, and third by annotating the first method only. As mentioned in the second obesrvation mentioned earlier, this experiment will show the effect of annotating nested methods

[3]Refer to Section 2.5 on page 18 for more information about the Pacemaker System Specification [74].

in the call hierarchy on the upper-most calling method. Figure 5.5 shows the timing analysis for the upper-most method, `TransientDetectorScopeEntry.run()`, using a boxplot[4]. The central line in both boxes represents the median, while the hinges mark the quartiles and the whiskers are each up to 1.5x the interquartile range (IQR) from the closer quartile. The results show a clear overhead of using nested analysis. While the method of interest should take no more than 0.1ms, applying annotations to the methods in the call hierarchy of this method increased the worst case execution time of this method to about 0.2ms.

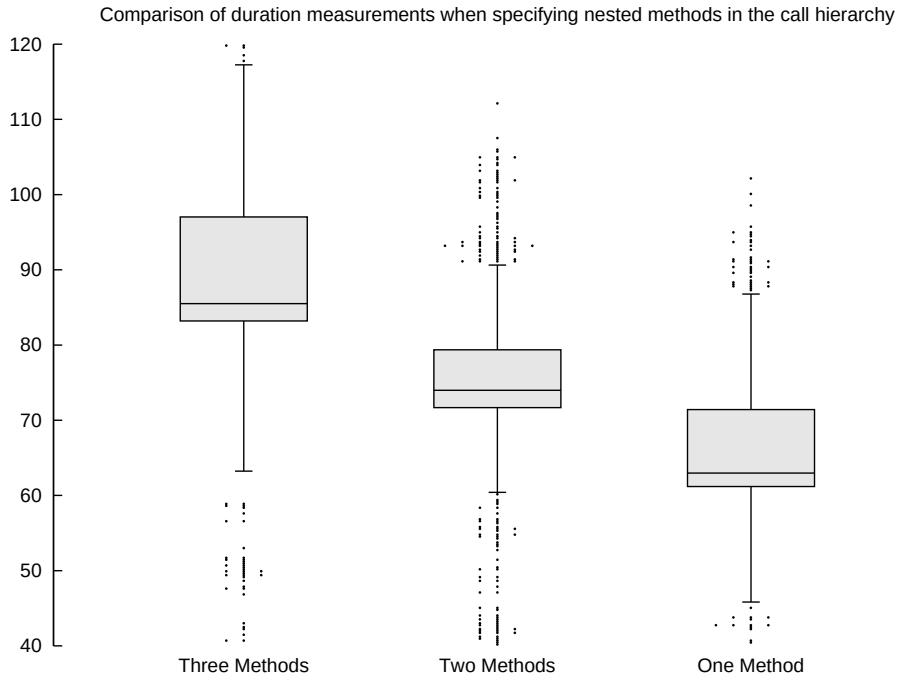

Figure 5.5: Comparison of `duration` clause timings for nested methods in a call hierarchy

One more observation from this evaluation is the fact that there are many outliers that are present in both experiments. In regression analysis, a data point is called an "outlier" if it diverges greatly from the overall pattern of data. This is represented in the figure by the individual points showing above and below the box whiskers. In this case, the difference in the maximum outlier between the first and the third experiments is about 0.01ms. This can be caused by other processes running on the machine and sharing the same hardware resources with the experiment is using. Therefore causing unpredictable delays in the execution. In fact, running the same experiment on dedicated hardware will eliminate the aforementioned reasons and result in more accurate measurements.

As shown earlier, by using SafeJML, the problem of timing analysis becomes a feature in the language and the programmer will no longer have to implement code to measure

[4]The boxplot is a statistical graph that displays a five number summary, these are: the sample minimum, the lower quartile or first quartile, the median, the upper quartile or third quartile and the sample maximum

durations. Moreover, timing methods and performing duration analysis becomes an easy task that is supported by the compiler.

The evaluation shows that the programmer can specify duration requirements by using the `duration` clause introduced earlier.

5.3 Evaluating the Duration Clause for Methods

In this section, experiments to evaulate the system for precision, direct feedback, and developer efficiency are presented. The SafeJML Runtime Assertion Checker must be precise enough to be useful. Precise measurements imply consistent overhead of the tool. For this reason, this evaluation measures the overhead of using the SafeJML analysis. The overhead of the measurements describe how imprecise the system is. Or in other words, this decides whether the SafeJML is "safe" to use or not. In this context, "safe" means that the analysis will not cause violations in the system in places where there should be be no violations. Based on measurements, users can determine whether an overhead is acceptable as an error margin in their timing analysis, or whether the analysis is considered "safe" for that program.

SafeJML must also give correct feedback to the programmer. Feedback is related to violations of timing constraints being related directly to the smallest specified units, that is the methods (or statements, in the case of refining statements) being specified. The evaluation shows that the error messages generated by the tool are actually useful for the user to identify the specific method that is violating timing specifications.

Last, SafeJML must be efficient for developers to use, in the sense that it adds only a small amount of overhead to the edit-compile cycle of a developer. Efficiency will be quantified by measuring the overhead of using the system versus using the basic FijiVM compiler.

5.3.1 Evaluating Precision

Precision is the degree to which repeated measurements under unchanged conditions show the same results. In this case, the measurement tool is the SafeJML duration runtime checker, and the measurements are the method duration.

To evaluate the imprecision of the tool, the same method used in Section 5.2 on page 56 was used. The experiment was modified to measure the overhead of using the duration clause versus eliminating the duration clause. This was done by manually inserting code that will record timestamps to measure the time the tool takes to record the duration

records. The experiment was then executed twice, (again, the experiment executes the method 1000 times each time it is invoked), once with SafeJML duration analysis enabled, and another time with the duration analysis disabled. Figure 5.6 shows the comparison between the two experiments using a boxplot.

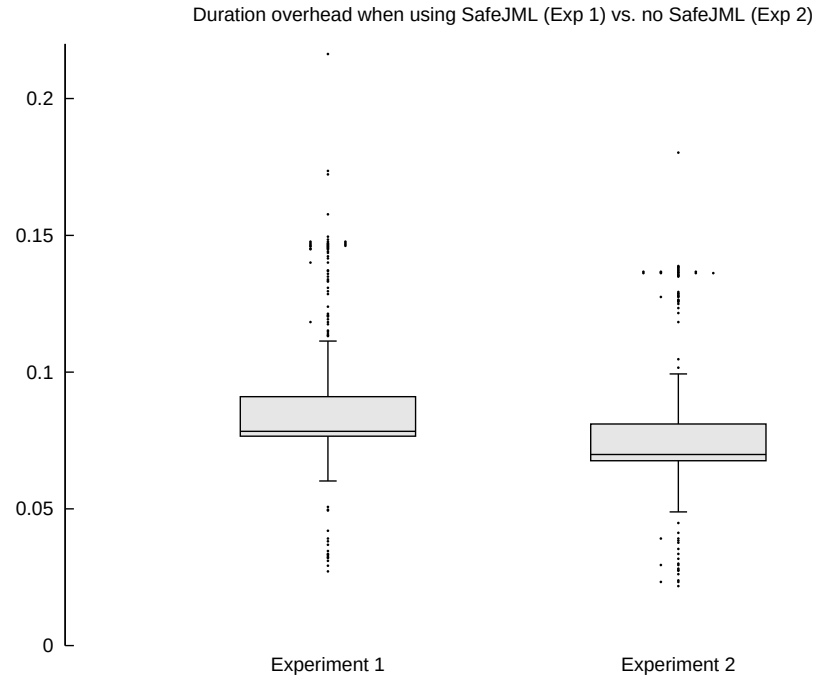

Figure 5.6: Comparison of duration of a method execution when the `duration` clause is used (Experiment 1) vs. not using the `duration` clause (Experiment 2) for the method `TransientDetectorScopeEntry.run()`

The graph shows a clear overhead when using the SafeJML tool to analyze the program. Recall that a memory-based approach is used to collect the datapoints for this experiment. A closer look at the quartiles of both experiments show some interesting facts. The value of the first quartile (Q1) of experiment 1 is about 0.077ms, while the value of the third quartile (Q3) is about 0.091ms. Q1 represents the 25th percentile of the data points, while Q3 represents the 75th percentile of the data points. In contrast, Q1 and Q3 for experiment 2 are 0.067ms and 0.081ms respectively. The interquartile range for both experiments is very close, being 0.014ms and 0.013ms for experiments 1 and 2 respectively. This means that 50% of the data points are bound in an almost equal interquartile range for both experiments. This indicates that the method of interest took, in both experiments, almost the same time to execute, that is if the overhead of the analysis is ignored. This is desirable as it indicates that the overhead of the analysis is close to being a constant value. However, the median in experiment 1 is clearly greater than in experiment 2. While the median in experiment 2 is about 0.070ms, it is around 0.078ms in experiment 1. The difference

between the two medians represents the median overhead for using the SafeJML tool for timing analysis. However, one should realize that there may be some additional imprecision in this measurement, although it does indicate that the overhead of using the SafeJML tool is about 0.008ms. This imprecision would be caused by the measurement tool itself.

Outliers can also be seen in this experiment. In this case, experiment 1 data extends to reach about 0.216ms at one point, while experiment 2 shows a maximum value of about 0.180ms. These extremes show a difference in duration measurements in the two experiments. This difference is an indication of external processes affecting the experimental setup. However, there is an upper bound to these measurements that the experiment never exceeded throughout the evaluation period. That is the one mentioned above. Therefore, statistical methods similar to the one presented earlier can determine whether it is safe to use SafeJML duration analysis for a particular program, or whether the analysis is considered "safe" for that program.

5.3.2 Evaluating Precision for Subtypes

To evaluate the precision of the tool and its ability to analyze duration specifications of subtypes, we studied the benchmark to find a good candidate type that can be used for the evaluation. After going through the benchmark, we selected a particular class as a good candidate to measure the tool's overhead. The class `Motion` in the benchmark describes a motion of an airplane using two 3D vectors that comprise the start and finish locations of a single airplane at a certain instance of the analysis. The class contains the method `findIntersection`, which provides the ability to detect whether two motions will intersect within a predefined radius. The method implements a complex algorithm that takes into consideration the speed of each airplane during the analysis, which the reader will recall is taking place every 50ms.

We believe that the algorithm found in `Motion`'s `findIntersection` method is more complex than necessary, as it does not take into consideration that the sampling rate of the algorithm is too fast so that any known aircraft cruising at its maximum speed cannot travel more than about 114 meters (374 feet) during the 50ms sampling rate.[5] Given this fact, we implemented an approximation to the algorithm for calculating collisions. The approximation was implemented as a supertype for the `Motion` type, named `ApproxMotion`. Using this implementation, the programmer can use the approximation algorithm in most cases, and only needs to use the subtype `Motion`'s algorithm for more accurate results. We also annotated the `findIntersection` method in both types using a `duration` clause. Figure 5.7 on the following page and Figure 5.8 on page 64 show the two annotated methods.

[5] The X-15 traveled at Mach 6.7, which is about 6629 feet per second, or about 374 feet per 50 ms. The Boeing 747, at 600 mph maximum speed, can travel around 44 ft in 50 ms.

The experiment was then run three times, at first, the original code `Motion.java` was annotated and duration data was collected. The code was then modified such that the original class became a subtype for the new supertype `ApproxMotion.java`. The two types were used to collect data for the second and third experiments, once by using the supertype, and the second time using the subtype. Duration records were collected and analyzed.

Figure 5.9 on page 65 shows a comparison of the three experiments using a boxplot. The y axis is the execution duration for each method in microseconds (μs). The graph shows an almost identical execution duration pattern for the first and last experiment, since these implement the same algorithm, it is expected for both executions to perform the same. This result proves that dynamic dispatch has minimal overhead on execution time, as in the third experiment, we are using a subtype and adding duration annotations in the supertype. The graph also shows a clear decrease in the method duration when using the introduced approximation compared to using the original implementation. It also shows that the tool was precise enough to detect this decrease.

One more observation from this evaluation is the fact that there are many outliers that are present in both experiments, as all three experiments extend to reach about 35μs. This indicates that while the implementation is straightforward, with very limited number of execution paths, the execution can suffer from inconsistencies in its duration due to its operating environment. However, these inconsistencies had upper bounds that were below the specified duration.

```
1  /*@  public normal_behavior
2    @    assignable \nothing;
3    @*/
4  public Vector3d findIntersection(final Motion other) {
5    // A complex algorithm that returns the intersection between
6    // the current motion and the motion 'other'
7  }
8  /*@
9  public pure model long findIntersectionTime() {
10   return 50 * org.jmlspecs.lang.DurationConstants.MICROSEC;
11 }
12 @*/
```

Figure 5.7: The method `findIntersection()` in the subtype `Motion`.

```
/*@  public normal_behavior
  @    requires other != null;
  @    ensures pos_one == \old(pos_one);
  @    ensures pos_two == \old(pos_two);
  @    duration 0 .. findIntersectionTime();
  @*/
public Vector3d findIntersection(final Motion other) {
  final float apr = Constants.PROXIMITY_RADIUS;
  final float xa0 = getFirstPosition().x - apr;
  final float xa1 = getSecondPosition().x + apr;
  final float xb0 = other.getFirstPosition().x - apr;
  final float xb1 = other.getSecondPosition().x + apr;
  final float ya0 = getFirstPosition().y - apr;
  final float ya1 = getSecondPosition().y + apr;
  final float yb0 = other.getFirstPosition().y - apr;
  final float yb1 = other.getSecondPosition().y + apr;
  final float za0 = getFirstPosition().z - apr;
  final float za1 = getSecondPosition().z + apr;
  final float zb0 = other.getFirstPosition().z - apr;
  final float zb1 = other.getSecondPosition().z + apr;
  if( (xa0 <= xb1 && xb0 <= xa1) &&
      (ya0 <= yb1 && yb0 <= ya1) &&
      (za0 <= zb1 && zb0 <= za1)) {
        return getFirstPosition();
  } else {
      return null;
  }
}
/*@
public pure model long findIntersectionTime() {
  return 35 *  org.jmlspecs.lang.DurationConstants.MICROSEC;
}
@*/
```

Figure 5.8: The method `findIntersection()` in the introduced supertype `ApproxMotion` added to the minicdx benchmark which shows the approximation for the algorithm.

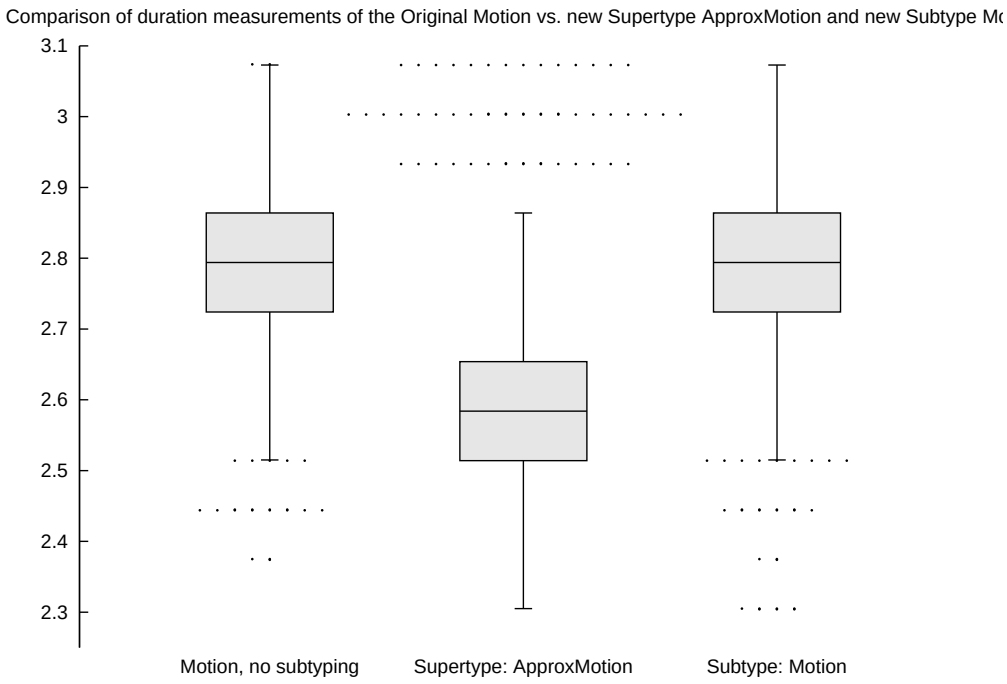

Figure 5.9: Comparing duration data for the original type `Motion`, the new supertype `ApproxMotion`, and the new subtype `Motion`, using SafeJML duration clause

5.3.3 Evaluating Feedback

In this section, I show how the SafeJML tool can provide correct and accurate feedback to the programmer by providing a timing report that is useful for the programmer to identify the specific method that is violating one or more timing specifications.

In order to evaluate the feedback of the tool, the duration specification for the method `findIntersection` shown earlier in Figure 5.8 on page 64 was set to finish no earlier than 17.5μs and no later than 120.0μs, and the same experiment that was used in Section 5.3.1 was used for this evaluation. Other methods were also annotated to provide more feedback. The tool then generated a report indicating that at least one instance of the run took more than 120.0μs to conclude. Similarly, at least one instance of the run took less than 17.5μs to conclude. Figure 5.10 on the following page shows the report generated by the duration runtime assertion checker.

This experiment shows that the tool gives direct feedback to the programmer by reporting the method name and the specification case that is violating the duration annotation.

5.3.4 Evaluating Developer Efficiency

Developer efficiency describes the extent to which time or effort is well used for the intended task or purpose. it is a measurable concept, quantitatively determined by the ratio of the outcome of the task compared to the effort spent to achieve this outcome. In the context of this evaluation, we identify the development efficiency of the tool by the time it takes for the developer to engage the tool for the purpose of measuring the duration of a method to execute. In other words, efficiency is measured by the overhead that the tool imposes on both compilation and post execution analysis. For the tool to be efficient to the developer, the tool must cause minimal usage overhead.

An experiment was conducted using a computer clock to measure the time it takes for the full cycle of the experiment to execute. The experiment was repeated once using the regular FijiVM and another time using the SafeJML tool chain. Table 5.1 shows efficiency analysis for both experiments. The results show about 7% overhead of the total execution time when SafeJML is used. However, the compiler itself generated very little overhead, and significant amount of time is spent during FijiVM compiler execution. It is important to mention here that runtime and analysis time depend on the number of runs per experiment, which is 1000 runs for this case. The number of runs are kept the same to measure the developer efficiency in a typical example.

```
Duration violations found...
============================

Maximum Duration Violations:
====================
Method{spec}                                  Duration Spec
============                                  ======== ====
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet
    {specCase_1}                              122153   [17500 .. 120000]
cdx.Motion.findIntersection{specCase_1}  22699    [1500 .. 10000]

Minimum Duration Violations:
====================
Method{spec}                                  Duration Spec
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet
    {specCase_1}                              16134    [17500 .. 120000]
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet
    {specCase_1}                              15854    [17500 .. 120000]
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet
    {specCase_1}                              17390    [17500 .. 120000]
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet
    {specCase_1}                              12921    [17500 .. 120000]
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet
    {specCase_1}                              17111    [17500 .. 120000]
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet
    {specCase_1}                              17460    [17500 .. 120000]
cdx.Motion.findIntersection{specCase_1}  1466     [1500 .. 10000]
cdx.Motion.findIntersection{specCase_1}  1257     [1500 .. 10000]
cdx.Motion.findIntersection{specCase_1}  1467     [1500 .. 10000]
cdx.Motion.findIntersection{specCase_1}  1466     [1500 .. 10000]
cdx.Motion.findIntersection{specCase_1}  1466     [1500 .. 10000]

Maximum execution times for this run...
=======================================
cdx.TransientDetectorScopeEntrySubtype.reduceCollisionSet->122153 ns
cdx.Motion.findIntersection->22699 ns
```

Figure 5.10: Report showing duration specification violation generated by the SafeJML Duration RAC.

Table 5.1: SafeJML Efficiency Analysis

Toolchain	**Stage**	**Duration (sec)**	**Duration (%)**
FijiVM	javac	6	2.4%
	Fiji Compiler	197	77.9%
	Runtime	50	19.8%
	Analysis	0	0.0%
	Total	253	100.0%
FijiVM + SafeJML	SafeJML Java Compiler	7	2.6%
	Fiji Compiler	200	73.8%
	Runtime	50	18.5%
	Analysis	14	5.2%
	Total	271	100.0%
	Overhead	18	7.2%

CHAPTER 6
CONCLUSIONS

This chapter presents my conclusions. First I present a summary for the work completed. A list of limitations and future work follows, then I present the contributions of this work.

6.1 Summary

Designers and developers of Safety-critical systems are becoming more interested in Java as a development tool. However, Java for safety-critical systems has very little support for specification languages and dynamic checking tools. In this dissertation, I presented SafeJML, a language that extends the support of JML as a specification language to include timing constraints and dynamic checking for systems developed using Safety-Critical Java (SCJ). Besides supporting dynamic checking, the language includes features that could be utilized in static checkers for timing constraints, used mainly in calculating worst case execution times for methods (WCET). I also presented SafeJML Duration Runtime Assertion Checker (RAC), a tool used to perform dynamic checking of timing constraints. The new tool was evaluated based on four requirements: expressiveness, precision, direct feedback, and developer efficiency. Furthermore, I discussed a practical problem related to using subtypes along with modular specifications. I also described a methodology to solve this problem, which enables the usage of modular specifications for timing constraints along with subtypes. This solution's implementation and evaluation was presented.

6.2 Limitations and Future Work

The work presented in this dissertation has the following limitations:

- No implementation is made available for a static verification tool that uses other SafeJML features to statically verify duration specification and perform WCET analysis.

- The current RAC implementation requires all methods to terminate normally, no exceptional termination is allowed. A future enhancement to overcome this limitation by using a `try-finally` statement is planned.
- The current RAC implementation might produce false positive results if used to analyze nested method calls. A future enhancement to account for the analysis overhead in nested method calls is left for future work.
- The current RAC reporting mechanism for methods violating their duration specifications does not include file name and line numbers. A future enhancement will be provided to support this detailed reporting.

6.3 Contributions

The following list contains the list of contributions of this work:

- Designed and implemented SafeJML, a modular specification language for Safety-critical systems built using Safety-Critical Java (SCJ).
- Designed and implemented the SafeJML Runtime Assertion Checking (RAC) tool. This tool dynamically checks duration specifications for methods and reports any violations for the programmer.
- Designed a methodology and algorithm to perform duration analysis for subtypes. The methodology was implemented as part of the SafeJML RAC tool.
- Designed and partially implemented JAJML, an extensible JML compiler. The SafeJML and SafeJML RAC implementations both used JAJML as their base implementations.

APPENDIX A
THE DESIGN OF JAJML

This appendix describes the approach used for building an extensible compiler for JML[1]. An example to build a runtime assertion checker for JML is also discussed. The compiler is based on the JastAdd attribute grammar tool [102, 103], and hence called JAJML. The developers of JastAdd have provided an extensible Java compiler, upon which JAJML is built. JAJML was in turn extended to support the features discussed throughout this work, or SafeJML.

The original work from which this appendix is based on, uses loop annotations to show the approach to building JAJML. For relevance, I decided to use the duration clause in SafeJML as an example for this appendix. In essence, the ideas used in both examples are the same. The reader is advised to refer to our technical report [101] for a broader discussion and the implementation of loop annotations in JAJML[2].

A.1 Building the JAJML Compiler

In this section, I use an example to describe how the duration clause was implemented in JAJML using the JastAdd compiler. As a first step, it is important to give a brief overview of JastAdd and then discuss the various tasks needed to make an extension: scanning, abstract syntax tree declaration, parsing, static analysis, and code generation.

The JastAdd Compiler Construction System [103, 102] extends Java with rewritable circular reference attribute grammars. In addition, JastAdd has other mechanisms like static aspect-oriented programming, declarative attributes, and context-dependent rewrites, that support the tool's modularity and extensibility. These techniques make writing extensible compilers more efficient. The acronym "JAJC" refers to the JastAdd extensible Java compiler, and "JastAdd" is used for the JastAdd compiler construction tool.

A.1.1 Scanning

The JAJC uses Flex for lexical analysis. Thus, in order to extend the JAJC, it is necessary to use Flex for lexical analysis of JAJC.

The scanner for JAJC is split into several files. Each file contains part of the overall Java lexical grammar, organized to ease extension. For example, the lexical grammar for comments is found in a file `Comments.flex`.

[1]This appendix is based on our technical report "Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations" [101].

[2]Sources and other information for the JAJML compiler can be obtained from `http://sourceforge.net/apps/trac/jmlspecs/wiki/JAJML`.

Flex has a feature that is very valuable for JAJC, namely the ability to use states to control the lexical analysis. JML specifications, and therefore, JAJC specifications, are contained within special annotation comments of the form `/*@ ... @*/` or from `//@` to the end of a line. Thus the lexical grammar uses several states to make sure it only recognizes JML keywords within such annotation comments [67, Section 4]. The addition of these states is done in a separate file, `JAJML.flex`, without editing or changing the JAJC's lexer files.[3].

A.1.2 Parsing

The JAJC uses the Beaver LALR(1) parser generator for parsing. Beaver accepts grammars expressed in EBNF and is well-integrated into JastAdd. This integration makes it easy to use Beaver's AST nodes, which are represented as instances of generated Java classes when manipulating attributes from within JastAdd.

The parser for JAJC is split over several files, as Beaver allows new context-free rules to be added to the grammar. This makes it easy to extend the grammar simply by adding new productions in separate files. For example, one can add productions for the duration expression by adding productions for the statement nonterminal. These productions can be found in the file `jajml.parser`. An example of the added code is shown in Figure A.1 on the next page. (This grammar follows the one introduced in Section 3.2.3.2 on page 30, which follows the same rules for JML grammar introduced in *JML Reference Manual* [67, Section 12.2].) In Beaver semantic actions are enclosed in `{:` and `:}` brackets. Each rule is responsible for creating and returning the corresponding AST node. Each AST node type is defined in the abstract grammar files as described above.

A.1.3 The Abstract Grammar

JastAdd supports a concise "abstract grammar" for declaring abstract syntax trees (ASTs). All classes that compose the AST are defined in abstract grammar files. JastAdd automatically generates Java classes to represent these ASTs.

The JAJC uses several abstract grammar files to define ASTs for Java. (There is one file for the Java 1.4 features and a file for each new feature added in Java 5.) JastAdd makes extending the hierarchy with new AST classes easy. This is done by adding new rules to the abstract grammar.

[3] However, flex does require duplicating the standard Java lexer's definitions within the new states.

```
JmlDuration duration_clause =
    JMLduration expression.e1 DOTDOT expression.e2 IF expression.p
       SEMICOLON
        {: return new JmlDuration(false, e1 ,e2 , new Opt(p)); :}
  | JMLduration_redundantly expression.e1 DOTDOT expression.e2 IF
     expression.p SEMICOLON
        {: return new JmlDuration(true, e1 ,e2 , new Opt(p)); :}
  | JMLduration expression.e1 DOTDOT expression.e2 SEMICOLON
        {: return new JmlDuration(false, e1 ,e2 , new Opt()); :}
  | JMLduration_redundantly expression.e1 DOTDOT expression.e2
     SEMICOLON
        {: return new JmlDuration(true, e1 ,e2 , new Opt()); :}
  | JMLduration BS_not_specified SEMICOLON
        {: return new JmlDuration(false, null, null, new Opt()); :}
  | JMLduration_redundantly BS_not_specified SEMICOLON
        {: return new JmlDuration(true, null, null, new Opt()); :}
  ;
;
```

Figure A.1: Part of the added grammar describing the duration clause.

```
abstract JmlStmt:Stmt;
abstract JmlAnnotationStmt: JmlStmt;
```

Figure A.2: AST for Loop Annotations Added to the `jajml.ast` File.

The added abstract grammar file describes the ASTs for JML's loop annotation statements. The declaration of the two AST classes that will serve as superclasses for other ASTs and will help organize them in the file `jajml.ast` are shown in Figure A.2.

The abstract AST class `JmlAnnotationStmt` inherits from the `JmlStmt` AST class, itself is an abstract AST class that inherits from the JAJC's `Stmt` AST class. For each feature, an entry in the `jajml.ast` file is added that declares the ASTs for that feature. ASTs for duration clause are shown in Figure A.3 on the following page. The AST class `JmlMethodSpec` is defined as a subclass of `JmlAnnotationStmt` which is shown earlier. The AST class `JmlDuration` is a subclass of `JmlMethodSpec`. The above says that a `JmlDuration` contains three nodes: the first two nodes are `First` and `Second`, both of type `Expr` and are used to hold the minumim and maximum duration constraints in a duration specification statement, and a `Pred` that is of type `Expr`, and is specified as `Optional`, hence the enclosing

square brackets. This definition of the `JmlDuration` AST can be directly mapped to the grammar introduced earlier in Figure A.1 on the previous page.

```
JmlMethodSpec:JmlAnnotationStmt ::= <IsRedundant:boolean>;
JmlDuration:JmlMethodSpec ::= First:Expr Second:Expr [Pred:Expr];
```

Figure A.3: AST for `duration` Clause Added to the `jajml.ast` File.

A.1.4 The Attribute Grammar

After parsing, and before applying rewrites and tree transformations, JastAdd compilers typically perform type checking and other static analysis. This allows messages about any problems in the user's program to be generated from unmodified ASTs that directly reflect the user's input and are thus easy for users to understand.

Type checking and other kinds of static analysis are performed using JastAdd's attribute grammar facilities. Attribute grammars attach attributes to ASTs. Attributes can be defined in either declarative or imperative style. While imperative style allows for writing regular Java code to manipulate the nodes, declarative style simply specifies the relations that define each attribute. Attributes are evaluated in an order determined automatically by JastAdd. Imperative style attribute definitions are executed when their values are needed by the declarative style attributed definitions, which controls the overall order of execution. Imperative style attribute definitions are written in Java code, and are useful when making complex decisions, for example in transformation code.

Attributes are implemented inside JastAdd aspects and can be either inherited or synthesized. Inside an aspect, one can define new attributes and equations[4] that are added to the different ASTs. This approach supports modularity of features, as each feature is implemented in a single aspect. Each aspect can affect many different types of ASTs, which supports separation of concerns.

There are several Java analyses that must be implemented in a compiler based on the JAJC. These analyses they must be implemented by AST classes such as `JmlDuration`. For example, type checking for the duration clause is implemented in JAJC with the synthesized

[4]In Attribute Grammar, an equation defines an attribute, the right-hand side of the equation is an expression that uses other attribute values to define the value of the left-hand side attribute. From an OO perspective, attributes are represented by fields, and equations are represented by methods that are used to compute the fields [103, Section 3.1].

```
public void JmlDuration.typeCheck(){
   TypeDecl lmt1 = getFirst().type();
   TypeDecl lmt2 = getSecond().type();
   if (!lmt1.isLong() && !lmt1.isInt()) {
      error("the type of \"" + getFirst() + "\" is " + lmt1.name()
         + ", but should be int or long");
   }
   if (!lmt2.isLong() && !lmt2.isInt()) {
      error("the type of \"" + getSecond() + "\" is " + lmt2.name()
         + ", but should be int or long");
   }
}
```

Figure A.4: Code from `duration.jrag`, for duration clause type checking.

Table A.1: Standard Java Analyses and Their Matching Attributes in JAJC

analysis	attribute
assignment checking	`isDAbefore(Variable)`
definite unassignment	`isDUafter(Variable)`
unreachable statements	`canCompleteNormally()`
variable lookup	`lookupVariable(String)`
name classification	`nameType()`
type checking	`typeCheck()`

attribute `JmlDuration.typeCheck()`. Figure A.4 shows the type checking attribute for the duration clause as implemented in the file `duration.jrag`.

Several other attributes needed to be defined and implemented for other SafeJML features in the same manner. Examples of these analyses are shown in Table A.1.

Several attributes were also added to extract parts of the JML duration clause ASTs, but all these are trivial.

A.1.5 Compiling Runtime Assertion Checks

This section describes how the prototype uses the facilities of JastAdd to compile runtime assertion checking code for JML duration clause[5]. Annotations in JAJML are

[5]Similar considerations would apply for such tasks as verification condition generation.

```
/*@ public behavior
  @ requires P;
  @ duration 300 .. 500;
  @ ensures Q;
  @*/
  T m(p1,p2,...) {
  // method body
}
```

Figure A.5: A method annotated with method duration specifications

```
T m(p1,p2,...) {
  checkPre(P,p1,p2,...);
  DurationRAC.enter(mID,Time());
  T ret = mWrapper(p1,p2,...)
  DurationRAC.exit(mID,Time());
  checkPost(Q,p1,p2,...)
  return ret;
}
T mWrapper(p1,p2,...) {
  // method body
}
```

Figure A.6: A transformed form of the annotated method

implemented by transforming the AST to include the original code woven with assertion checking code. Assertion checking code throws JML-specific error objects when an assertion fails.

JAJML uses assertions derived from duration clause to insert `Enter` and `Exit` methods for the purpose of timing the method, as discussed in Section 3.3.1 on page 36. These methods will then record method entry and exit timing information during execution.

An annotated method is considered to include the entry and exit timings. To explain this precisely, The process in which the annotated JML source code is transformed into Java with runtime assertion checks is specified. This is done by transforming the method to include these calls. Code of the form shown in Figure A.5, can be transformed to the form shown in Figure A.6.

The prototype example expresses this kind of transformation directly at the level of abstract syntax trees in JastAdd. The prototype transforms the method to include entry and exit calls and passes the method ID as discussed in Section 4.1 on page 44.

For method transformation, JastAdd's `transformation()` function was used. The choice was to made in preference to JastAdd's `Rewrite` construct, because transformations are done after the static analysis steps described above, and this, as described above, generally leads to better error message generation. This transformation function is implemented for `JmlDuration`.

LIST OF REFERENCES

[1] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Addison Wesley Longmain, 3 ed., 2001.

[2] N. Leveson, *Safeware : System Safety and Computers*. Reading, Mass.: Addison-Wesley Pub Co., 1995.

[3] P. G. Neumann, "The risks digest." `http://catless.ncl.ac.uk/Risks`.

[4] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, pp. 23 –29, oct 2000.

[5] A. Armbuster, J. Baker, A. Cunei, D. Holmes, C. Flack, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek, "A Real-time Java virtual machine with applications in avionics," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, pp. 1–49, Dec. 2007.

[6] "Boeing selects software for j-ucas x-45c," *Defense Industry Daily*, Oct. 2005.

[7] "The jamaicavm brings java technology to mission software in an unmanned aircraft by eads," *Military Embedded Systems*, June 2006.

[8] C. Hammerschmidt, "Test platform connects oems and tier-ones," *EE Time Europe*, Oct. 2007.

[9] F. de Bruin, F. Deladerrire, and F. Siebert, "A standard java virtual machine for real-time embedded systems," *DASIA'03, Prague*, 2003.

[10] N. Juillerat, S. Müller Arisona, and S. Schubiger-Banz, "Real-time, low latency audio processing in java," in *Proceedings of the International Computer Music Conference*, (Copenhagen, Denmark), Aug. 2007.

[11] S. Gestegard Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, "Using real-time Java for industrial robot control," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES)*, pp. 104–110, 2007.

[12] "Aonix PERC selected for inflight entertainment system," *Embedded Computing Design*, Sept. 2007.

[13] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, pp. 212–232, June 2005.

[14] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond assertions: Advanced specification and verification with JML and ESC/Java2," in *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, vol. 4111 of *Lecture Notes in Computer Science*, (Berlin), pp. 342–363, Springer-Verlag, 2006.

[15] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, pp. 1–38, Mar. 2006.

[16] G. T. Leavens, "JML's rich, inherited specifications for behavioral subtypes," in *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)* (Z. Liu and H. Jifeng, eds.), vol. 4260 of *Lecture Notes in Computer Science*, (New York, NY), pp. 2–34, Springer-Verlag, Nov. 2006.

[17] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman, "JML Reference Manual." Available from `http://www.jmlspecs.org`, Sept. 2009.

[18] J. M. Wing, "Writing Larch interface language specifications," *ACM Trans. Programming Languages and Systems*, vol. 9, pp. 1–24, Jan. 1987.

[19] J. Krone, W. F. Ogden, and M. Sitaraman, "Modular verification of performance correctness," in *ACM OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pp. 60–67, 2001.

[20] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek, "Cdx: a family of real-time java benchmarks," in *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, (New York, NY, USA), pp. 41–50, ACM, 2009.

[21] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Proc. First International Workshop on Embedded Software (EMSOFT 2001)*, vol. 2211 of *Lecture Notes in Computer Science*, pp. 469–485, Springer-Verlag, 2001.

[22] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proceedings of the IEEE*, vol. 91, pp. 1038–1054, July 2003.

[23] R. Heckmann and C. Ferdinand, "Worst-case execution time prediction by static program analysis." `http://www.absint.com/aiT_WCET.pdf`, 2006.

[24] G. Haddad and G. T. Leavens, "Specifying subtypes in SCJ programs," in *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, (New York, NY, USA), pp. 40–46, ACM, 2011.

[25] J. Gustafsson, "Worst case execution time analysis of object-oriented programs," (Los Alamitos, CA, USA), p. 0071, IEEE Computer Society, 2002.

[26] G. T. Leavens and K. K. Dhara, "Concepts of behavioral subtyping and a sketch of their extension to component-based systems," in *Foundations of Component-Based Systems* (G. T. Leavens and M. Sitaraman, eds.), ch. 6, pp. 113–135, Cambridge, UK: Cambridge University Press, 2000.

[27] G. T. Leavens and D. A. Naumann, "Behavioral subtyping, specification inheritance, and modular reasoning," Tech. Rep. 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Sept. 2006.

[28] G. T. Leavens and W. E. Weihl, "Specification and verification of object-oriented programs using supertype abstraction," *Acta Informatica*, vol. 32, pp. 705–778, Nov. 1995.

[29] K. R. M. Leino, "Recursive object types in a logic of object-oriented programs," *Nordic Journal of Computing*, vol. 5, pp. 330–360, Winter 1998.

[30] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck, and R. Trummer, "Java takes flight: time-portable real-time programming with exotasks," *ACM SIGPLAN Notices*, vol. 42, no. 7, pp. 51–62, 2007.

[31] J. Gosling and H. McGilton, "The Java language environment: A white paper," tech. rep., Sun Microsystems, June 1996.

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

[33] D. C. Sharp, "Real-time distributed object computing: Ready for mission-critical embedded system applications," in *Proceeding of the 3rd International Symposium on Distributed Objects and Applications, DOA 2001, 17-20 September 2001, Rome, Italy*, (Silver Spring, MD 20910, USA), pp. 3–4, IEEE Computer Society Press, 2001.

[34] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz, "Project Golden Gate: Towards Real-Time Java in Space Missions," in *Proceedings of the 7th IEEE International Symposium*

on Object-Oriented Real-Time Distributed Computing (ISORC), 12-14 May 2004, Vienna, Austria, (Silver Spring, MD 20910, USA), pp. 15–22, IEEE Computer Society Press, 2004.

[35] K. Nilsen, "Using java for reusable embedded real-time component libraries," *CrossTalk: The Journal of Defense Software Engineering*, vol. 17, pp. 13–18, Dec. 2004.

[36] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain, "Mackinac: Making hotspot real-time," in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pp. 45–54, 2005.

[37] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. Stoodley, "Design and implementation of a comprehensive real-time Java virtual machine," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT)*, pp. 249–258, 2007.

[38] Aonix, "Perc pico 1.1 user manual." http://research.aonix.com/jsc/pico-manual.4-19-08.pdf, April 2008.

[39] aicas, "The Jamaica Virtual Machine homepage, http://www.aicas.com," 2005.

[40] "Lockheed martin selects aonix perc virtual machine for aegis weapon system," *Military Embedded Systems*, Oct. 2006.

[41] B. McCloskey, D. Bacon, P. Cheng, and D. Grove, "Staccato: A parallel and concurrent real-time compacting garbage collector for multiprocessors," research report, IBM, Feb. 2008.

[42] S. R. Schach, *Object-Oriented and Classical Software Engineering*. New York, NY: McGraw-Hill, 2005.

[43] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*. Cambridge, Mass.: The MIT Press, 1986.

[44] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing, *Larch: Languages and Tools for Formal Specification*. New York, NY: Springer-Verlag, 1993.

[45] G. T. Leavens and M. Sitaraman, eds., *Foundations of Component-Based Systems*. New York, NY: Cambridge University Press, 2000.

[46] Sun Microsystems, Inc., "JSR 302: Safety critical java technology." From `http://jcp.org/en/jsr/detail?id=302` (Date retrieved: March 19, 2008), 2007.

[47] Radio Technical Commission for Aeronautics (RTCA), *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, 1982.

[48] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek, "Java for safety-critical applications," *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[49] Purdue University - S3 Lab, "The Ovm Virtual Machine homepage, `http://www.ovmj.org/`," 2005.

[50] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek, "Engineering a customizable intermediate representation," in *Proceedings of the Workshop on Interpreters, Virtual Machines, and Emulators (IVME'03), San Diego, California*, 2003.

[51] C. Flack, T. Hosking, and J. Vitek, "Idioms in Ovm," Tech. Rep. CSD-TR-03-017, Purdue University Department of Computer Sciences, 2003.

[52] Computer-Science Department Annual Report, Purdue University, *oSCJ: Open Safety-Critical Java Project, White Paper*, January 2010.

[53] B. Meyer, *Object-oriented Software Construction*. New York, NY: Prentice Hall, second ed., 1997.

[54] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications ACM*, vol. 12, pp. 576–580,583, Oct. 1969.

[55] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software—Practice & Experience*, vol. 35, pp. 583–599, May 2005.

[56] G. T. Leavens and C. Clifton, "Lessons from the JML project," Tech. Rep. 05-12a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 2005. Available by anonymous ftp from ftp.cs.iastate.edu.

[57] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," in *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Lieden, The Netherlands, November 2002, Revised Lectures* (F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), vol. 2852 of *Lecture Notes in Computer Science*, pp. 262–284, Berlin: Springer-Verlag, 2003.

[58] K. Arnold and J. Gosling, *The Java Programming Language*. The Java Series, Reading, MA: Addison-Wesley, second ed., 1998.

[59] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. The Java Series, Reading, MA: Addison-Wesley, 1996.

[60] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, pp. 40–51, Oct. 1992.

[61] B. Meyer, *Eiffel: The Language*. Object-Oriented Series, New York, NY: Prentice Hall, 1992.

[62] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, pp. 8–24, Sept. 1990.

[63] R. J. R. Back, "A calculus of refinements for program derivations," *Acta Informatica*, vol. 25, pp. 593–624, Aug. 1988.

[64] R.-J. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science, Berlin: Springer-Verlag, 1998.

[65] C. Morgan and T. Vickers, eds., *On the refinement calculus*. Formal approaches of computing and information technology series, New York, NY: Springer-Verlag, 1994.

[66] C. Morgan, *Programming from Specifications: Second Edition*. Hempstead, UK: Prentice Hall International, 1994.

[67] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman, "JML Reference Manual." Available from `http://www.jmlspecs.org`, May 2008.

[68] Y. Cheon, "A runtime assertion checker for the Java Modeling Language," Tech. Rep. 03-09, Department of Computer Science, Iowa State University, Ames, IA, Apr. 2003. The author's Ph.D. dissertation.

[69] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, vol. 37(5) of *SIGPLAN*, (New York, NY), pp. 234–245, ACM, June 2002.

[70] D. R. Cok and J. Kiniry, "ESC/Java2: Uniting ESC/Java and JML," tech. rep., University of Nijmegen, 2004. NIII Technical Report NIII-R0413.

[71] L. Burdy, J.-L. Lanet, and A. Requet, "JACK: Java applet correctness kit," in *4th Gemplus Developer Conference*, Nov. 2002. `http://www.gemplus.com/smart/r_d/trends/jack.html`.

[72] B. Jacobs and E. Poll, "Java program verification at Nijmegen: Developments and perspective," Tech. Rep. NIII-R0318, Computing Science Institute, University of Nijmegen, 2003.

[73] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin, "JML Reference Manual." Available from `http://www.jmlspecs.org`, Oct. 2007.

[74] B. Scientific, "Pacemaker system specification," tech. rep., Boston Scientific, http://www.cas.mcmaster.ca/sqrl/_SQRLDocuments/PACEMAKER.pdf, January 2007.

[75] G. Haddad, F. Hussain, and G. T. Leavens, "The design of SafeJML, a specification language for SCJ with support for WCET specification," in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, (New York, NY, USA), pp. 155–163, ACM, 2010.

[76] R. Kirner, P. Puschner, and I. Wenzel, "Measurement-based worst-case execution time analysis using automatic test-data generation," in *Proc. 4th Euromicro International Workshop on WCET Analysis*, pp. 67–70, June 2004.

[77] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR*, (New Mexico State University), pp. 24–27, Jonathan Cook, May 2003.

[78] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek, "Developing safety critical java applications with oSCJ/L0," in *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, (New York, NY, USA), pp. 95–101, ACM, 2010.

[79] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, "High-level programming of embedded hard real-time devices," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, (New York, NY, USA), pp. 69–82, ACM, 2010.

[80] J. Krone, W. F. Ogden, and M. Sitaraman, "Profiles: A compositional mechanism for performance specification," Tech. Rep. RSRG-04-03, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, June 2004. Invited as one of the best papers from the SAVCBS Workshop series and under consideration for Formal Aspects of Computing, Springer-Verlag.

[81] S. M. Shaner, G. T. Leavens, and D. A. Naumann, "Modular verification of higher-order methods with mandatory calls specified by model programs," in *International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada*, (New York, NY), pp. 351–367, ACM, Oct. 2007.

[82] Y. Cheon and G. T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002* (H. R. Arabnia and Y. Mun, eds.), pp. 322–328, CSREA Press, June 2002.

[83] A. Shaw, *Real-Time Systems and Software*. New York, NY: John Wiley & Sons, 2001.

[84] M. Schoeberl and R. Pedersen, "WCET analysis for a java processor," in *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, (New York, NY, USA), pp. 202–211, ACM, 2006.

[85] E. C. R. Hehner, "Formalization of time and space," *Formal Aspects of Computing*, vol. 10, pp. 290–306, 1998.

[86] G. Bernat, A. Colin, and S. Petters, "pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems," in *Proc. 3rd Int. Workshop on WCET Analysis, Satellite Workshop of the Euromicro Conference on Real-Time Systems, Porto, Portugal*, July 2003.

[87] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution," *Real-Time Systems Symposium, IEEE International*, vol. 0, pp. 57–66, 2006.

[88] J. Gustafsson and A. Ermedahl, "Automatic derivation of path and loop annotations in object-oriented real-time programs," *Parallel and Distributed Real-Time Systems, Workshop*, vol. 0, p. 257, 1997.

[89] E. Y.-S. Hu, G. Bernat, and A. Wellings, "A static timing analysis environment using java architecture for safety critical real-time systems," *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, vol. 0, p. 0077, 2002.

[90] L. Cardelli and P. Wegner, "On understanding types, data abstraction and polymorphism," *ACM Computing Surveys*, vol. 17, pp. 471–522, Dec. 1985.

[91] G. T. Leavens, "Modular verification of object-oriented programs with subtypes," Tech. Rep. 90-09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from `ftp.cs.iastate.edu`, and by e-mail from almanac@cs.iastate.edu.

[92] G. T. Leavens, "Verifying object-oriented programs that use subtypes," Tech. Rep. 439, Massachusetts Institute of Technology, Laboratory for Computer Science, Feb. 1989. The author's Ph.D. thesis.

[93] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Programming Languages and Systems*, vol. 16, pp. 1811–1841, Nov. 1994.

[94] I. J. Hayes and M. Utting, "A sequential real-time refinement calculus," *Acta Informatica*, vol. 37, no. 6, pp. 385–448, 2001.

[95] M. Parkinson and G. Bierman, "Separation logic, abstraction and inheritance," in *ACM Symposium on Principles of Programming Languages* (P. Wadler, ed.), (New York, NY), pp. 75–86, ACM, Jan. 2008.

[96] M. J. Parkinson, "Local reasoning for Java," Tech. Rep. 654, University of Cambridge Computer Laboratory, Nov. 2005. The author's Ph.D. dissertation.

[97] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen, "Model-based schedulability analysis of safety critical hard real-time java programs," in *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, JTRES '08, (New York, NY, USA), pp. 106–114, ACM, 2008.

[98] M. Schoeberl, "A time predictable Java processor," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, (Munich, Germany), pp. 800–805, March 2006.

[99] E.-S. Hu, G. Bernat, and A. Wellings, "Addressing dynamic dispatching issues in wcet analysis for object-oriented hard real-time systems," in *Object-Oriented Real-Time Distributed Computing, 2002. (ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, pp. 109 –116, 2002.

[100] E. Hu, A. Wellings, and G. Bernat, "XRTJ: An extensible distributed high-integrity real-time java environment," in *Real-Time and Embedded Computing Systems and Applications* (J. Chen and S. Hong, eds.), vol. 2968 of *Lecture Notes in Computer Science*, pp. 208–228, Springer Berlin / Heidelberg, 2004.

[101] G. Haddad and G. T. Leavens, "Extensible dynamic analysis for jml: A case study with loop annotations," Tech. Rep. CS-TR-08-05, School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, Florida, April 2008.

[102] T. Ekman and G. Hedin, "The JastAdd system — modular extensible compiler construction," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 14–26, 2007.

[103] T. Ekman, *Extensible Compiler Construction*. PhD thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 2006. Dissertation 25.