Specifying Subtypes in SCJ Programs

Ghaith Haddad and Gary T. Leavens

CS-TR-11-04
July 2011

Keywords: safety-critical, real-time, WCET, timing analysis, timing constraints, timing behavior, formal specification, verification, oSCJ, RapiTime, Safety Critical Java (SCJ), SafeJML specification language.

2011 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools, JML; D.2.4 [*Software Engineering*] Software/Program Verification — Formal methods, programming by contract, reliability, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, logics of programs, pre- and post-conditions, specification techniques;

Submitted for publication.

School of EECS
4000 Central Florida Blvd.
University of Central Florida
Orlando, Florida 32816, USA

Specifying Subtypes in SCJ Programs

Ghaith Haddad
University of Central Florida,
Orlando, FL, USA
haddad@ieee.org

Gary T. Leavens
University of Central Florida,
Orlando, FL, USA
leavens@eecs.ucf.edu

ABSTRACT

Modular reasoning about programs that use subtypes requires that an overriding method in a subtype obeys the specifications of all methods that it overrides. For example, if method m is specified in a supertype T to take at most 42 nanoseconds to execute, then m cannot take more than 42 nanoseconds to execute in any subtype of T. Subtyping is an important aid to maintenance of programs, since it allows one to write polymorphic code (reducing code size and increasing reuse), and allows for convenient extension and enhancement of programs, all of which could be very useful in real-time programming. In this paper we show how to specify timing constraints for subtypes in a way that: permits modular reasoning about timing constraints, supports subtype polymorphism and object-oriented design patterns, and still permits precise reasoning about execution times. This technique supports object-oriented coding and design patterns based on subtype polymorphism, with all their maintenance advantages, to be used in real-time software.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.4 [**Performance of Systems**]: Measurement techniques, performance attributes; D.2.1 [**Software Engineering**]: Requirements/Specifications Languages, tools; D.2.4 [**Software Engineering**]: Software/Program Verification Assertion checkers,formal methods, programming by contract, validation; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about ProgramsAssertions, specification techniques

Keywords

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

1. INTRODUCTION AND MOTIVATION

Subtyping is an important coding technique for programming in OO languages. However, real-time programmers usually try to stay away from subtyping, due to the perceived complications for timing analysis. Consider the example in Figures 1 and 2. This code is written in Safety Critical Java (SCJ) [34]. Figure 1 declares a type `Vector2d` that represents two-dimensional vectors. Figure 2 shows another type, `Vector3d`, which is a subtype of `Vector2d`. To save space, we only show a representative method in these types, the `scale` method, which scales the dimensions of the vector by a floating point number `factor`.

```
public class Vector2d {

    protected /*@ spec_public @*/ float x, y;

  /*@   public normal_behavior
    @     requires !Float.isNaN(factor);
    @     assignable x, y;
    @     ensures x == \old(x) * factor
    @             & y == \old(y) * factor;
    @     duration 2 *
    @        (ITimeConstants.MultiplyTime
    @          + ITimeConstants.AssignTime); @*/
    public void scale(float factor) {
        this.x *= factor;
        this.y *= factor;
    }
}
```

Figure 1: 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, are parsed by the SafeJML checker.

The **`spec_public`** modifier makes the fields `x` and `y` public for specification purposes. The specification of the `scale` method precedes the header of that method. It includes a **`requires`** clause which specifies the method's precondition. The **`assignable`** clause specifies that the method is only allowed to assign to the `x` and `y` fields. The **`ensures`** clause specifies that the `x` and `y` fields are scaled by the given factor. The **`duration`** clause specifies the maximum time the method's execution may take. The example's duration clause specifies the method's maximum time using some constants that depend on the platform that this code is written for.

The subtype, `Vector3d` shown in Figure 2 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 specifications in this subtype `Vector3d` are similar to the ones that are in the supertype, and further restrict the specifications inherited from the supertype, `Vector2d`.

A standard methodology, *supertype abstraction*, for modular reasoning in the presence of subtype polymorphism, uses the specifications associated with each receiver's static type [19, 24] to reason

```
public class Vector3d extends Vector2d {

    protected /*@ spec_public @*/ float z;

  /*@ also
    @   public normal_behavior
    @     requires !Float.isNaN(factor);
    @     assignable z;
    @     ensures z == \old(z) * factor;    @*/
    public void scale(float factor) {
        super.scale(factor);
        this.z *= factor;
    }
}
```

Figure 2: `Vector3d`, a subtype of `Vector2d`. The specification of its `scale` method adds to the specification inherited from its supertype, `Vector2d`.

about method calls. Validity of supertype abstraction is guaranteed by *behavioral subtyping*, which requires that each subtype's overriding methods to obey the specification of each method that they override [19, 21, 22, 26]. In JML, as in our example, the subtype `Vector3d` is a behavioral subtype of its supertype, `Vector2d`, since JML uses specification inheritance [19] to force the subtype to obey the supertype's specification for the `scale` method. That is, whenever a call to `Vector3d`'s `scale` method satisfies the precondition of its supertype's `scale` method, then the implementation in `Vector3d` must also satisfy the frame condition and post condition specified in the supertype `Vector2d`. (This ensures that client reasoning about calls made on receiver objects of static type `Vector2d`, which use `Vector2d`'s specification, are valid even if the receiver's dynamic type is the subtype, `Vector3d`.)

However, applying behavioral subtyping to timing constraints poses a practical problem. As in our example, it is often the case 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` will not be able to scale the vector within the specified time, because the duration clause only accounts for two multiplication operations and two assignments, while the overridden method in `Vector3d` needs a third multiplication and assignment.[1]

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 [25]. (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).

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 we changed the specification in Figure 1 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 we imagined doing 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 standard OO libraries, which already use subtyping and method overriding extensively.

In this work, we try to 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. We also introduce a tool that we implemented to experiment with our ideas and demonstrate that our approach is feasible.

1.1 SafeJML

The Java Modeling Language (JML) [2, 3, 20, 19, 23] is a behavioral interface specification language [35] that boasts many state-of-the-art features for functional specification. Previously, we introduced *SafeJML* [8], an extension to JML for Safety-Critical Java (SCJ) programs. SafeJML is implemented as an extension to *JAJML* [9].

Besides allowing specification of functional behavior, SafeJML allows SCJ users to specify and check timing constraints for methods. SafeJML's **`duration`** clause is based on the clause with the same name found in JML, but revised to specify time in absolute time units (nanoseconds). SafeJML also has several other features that were added to enable the use of various tools to check timing constraints.

In SafeJML one can specify timing constraints for each method using the **`duration`** clause. We understand that real-time programmers do not typically specify timing constraints for each method. However, even if programmers do not wish to divide the time budget for a task so finely, the ability to specify timing constraints for each method will allow programmers to swiftly find which methods are causing timing problems. For example, method timing specifications can be added after the code is written and some timing measurements have been made, with the goal of discovering which methods are responsible for a task going over an expected end-to-end timing budget for a task or response. Furthermore, just as with functional specifications, timing specifications can serve as contracts, and thus can support division of labor and allow modular reasoning about timing constraints. As real-time systems become more complex, such aids to modularity will be more valuable.

SafeJML was originally designed [8] to be used with both static analysis tools such as AbsInt's aiT [6, 12, 11] and dynamic analysis tools such as RapiTime [14]. 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 [5]. Thus SafeJML is designed to support both kinds of analysis.

Currently we are focusing on dynamic analysis (runtime checking) for timing constraints. However, when we initially tried to use RapiTime for dynamic analysis, we discovered its limitations (as of this writing) on the size of C files it can process in a given run. This limitation caused problems for us, because of the way we are compiling SCJ to C code.

Compilation of SCJ code takes place in a SCJ implementation.

[1] Thus as specified so far, `Vector3d` cannot be correctly implemented. This is the way that problems with behavioral subtyping manifest themselves in JML, due to specification inheritance.

Such an implementation is provided by our partners at Purdue University (Jan Vitek and his group). This implementation is called oSCJ [4, 30]. It is an open-source SCJ implementation based on OVM [31]. Currently oSCJ implements all of Level 0 of SCJ [34]. oSCJ compiles SCJ code and the virtual machine, together, into C and then uses a standard C compiler. Lately, oSCJ has another implementation based on the Fiji VM [29]. However, both of these VMs produce large C source files, and that caused RapiTime to stop functioning. The Fiji VM is a Java virtual machine dedicated to embedded, hard real-time devices, and just like OVM, it compiles Java byte code to C. It also allows users to combine Java code with C code. Such C code is often used for device drivers and other low-level parts of safety critical systems.

To overcome the problems with RapiTime, we implemented a simplified analysis procedure 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.

1.2 Syntax and semantics of SafeJML

SafeJML, like JML, specifies methods using contracts ("specification cases") [23]. The clauses in such a contract are used to specify the method's behavior. These include the **`requires`**, **`assignable`**, and **`ensures`** clauses, which specify (respectively) the precondition, frame condition, and post condition of a method. For SafeJML, we are particularly interested in the *duration-clause*.

duration-clause ::= **`duration`** *spec-expression* **`;`**

The semantics of JML's duration clause is based on the work of Krone *et al.* [15, 16]. It specifies the maximum execution time needed for a method to execute in absolute time units.[2] 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.

2. SUBTYPING AND WCET ANALYSIS

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 we described in the introduction, some researchers in real-time systems suggest that this feature should be disallowed [7]. However, we will show how to solve these reasoning problems by using standard techniques for modular reasoning in the presence of subtype polymorphism, and show 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 and the use of object-oriented design patterns.

A standard methodology for modular reasoning in the presence of subtype polymorphism, called *supertype abstraction* [19, 24], 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 example, 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 post condition, taken from the static type of `o`, implies Q.)

[2] However, such a specification only applies when the precondition of the specification case in which it appears is satisfied [23, 19].

Soundness of supertype abstraction requires types to be behavioral subtypes of their supertypes [22]. If S is a subtype of T, then S is a *behavioral subtype* of T only if all overriding methods in S obeys their specification in T [18, 17, 19, 21, 22, 24, 26].

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 often 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, 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.[3]

As we explained in the introduction, 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.

2.1 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 [10, 15, 16]. As our examples 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 *dynamic type* of the receiver.

This insight dovetails with an elegant way to express the dependency of a method's specification on the dynamic type of the method's receiver, first published by Matthew Parkinson [28, 27]. The essence of Parkinson's technique is to write specifications of methods using "abstract predicate families" [27, p. 78], which can have differing definitions in various types. The value of an abstract predicate depends on the runtime type of a method's receiver. In SafeJML an abstract predicate family is declared as a non-static **`pure model`** method; such a model method can only be used in specifications, and yet can be overridden in different types, and thus can have a different meaning for each subtype. Thus, during checking (which takes place after running the program), one must know something about the runtime type of the receiver in order to use a particular member of the abstract predicate family to interpret a specification.

One may also specify various properties of abstract predicate families that allow reasoning with inexact type information (upper bounds). Parkinson's work gives sound rules for reasoning about specifications that are written using abstract predicates, which apply to SafeJML's use of model methods.

Consider the example we introduced earlier, 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();
```

[3] 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.

The method `scaleTime` would be overridden in each concrete type. 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`**). 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 the above 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 [24, 19, 22]) if desired. 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 that more exact type's `scaleTime` method.

2.2 Implementation of Our Solution

For runtime checking of timing constraints, it is important that the implementation consumes minimal (and constant) time. Our implementation does this by outputting an execution trace, during the program's execution, with enough timing information to enable later check of timing constraints. When the program is not executing, the tool checks the program's execution trace and compares the duration of methods with that the program's specified duration clauses.

Our implementation's design is shown in Figure 3. The analysis process consists of four stages. The first stage is the SafeJML compiler stage. The SafeJML compiler is a Java compiler with the ability to process JML specification. 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`) 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.

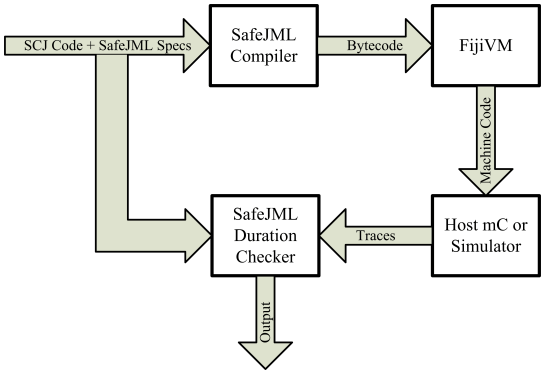

Figure 3: Design diagram

The Fiji compiler compiles class files into C code which in turn is compiled into machine code. The Fiji compiler takes care of this process internally. Then the generated machine code moves to the third stage, the host microcontroller. This microcontroller could be the actual target hardware or a simulator. Running the code will result in producing execution traces. These execution traces are processed by the SafeJML checker stage, which also takes the original 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.

The novel feature of our 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.

To record execution traces, the SafeJML compiler injects entry and exit method calls into code of each SafeJML-annotated method. The calls to the entry and exit methods are passed a parameter value that represents the SafeJML-annotated method's ID, and another parameter that represents the receiver's dynamic type. This information is encoded in the execution traces in a way that minimizes space and makes recovery of the information fairly easy.

Several methods to collect traces are supported. For example, if the code is going to be simulated on a PC, the programmer then will choose to collect traces in the memory 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 collected on different hardware. The net result must 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.

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. The checker must also use the dynamic receiver type information in the traces to figure out which overridden method was actually executed during runtime.

```
public class Vector2d {

    protected /*@ spec_public @*/ float x, y;

  /*@   public normal_behavior
    @     requires !Float.isNaN(factor);
    @     assignable x, y;
    @     ensures x == \old(x) * factor
    @             & y == \old(y) * factor;
    @     duration scaleTime();           @*/
    public void scale(float factor) {
        this.x *= factor;
        this.y *= factor;
    }

  /*@
    public pure model long scaleTime() {
        return this.getDimensions()
          * (ITimeConstants.MultiplyTime
              + ITimeConstants.AssignTime);
        }
    @*/

  /*@ ensures \result >= 2;
    public pure model int getDimensions() {
        return 2;
    }
    @*/
}
```

Figure 4: Specifications for `Vector2d`, modified from those in Figure 1 to use the proposed approach with model methods (following Parkinson *et al.*).

```
public class Vector3d extends Vector2d {

    protected /*@ spec_public @*/ float z;

  /*@ also
    @   public normal_behavior
    @     requires !Float.isNaN(factor);
    @     assignable z;
    @     ensures z == \old(z) * factor;    @*/
    public void scale(float factor) {
        super.scale(factor);
        this.z *= factor;
    }

    /*@
    public pure model int getDimensions() {
        return 3;
    }
 @*/

}
```

Figure 5: Specifications for `Vector3d`, modified from those in Figure 2 to use the proposed approach.

We show our proposed solution in Figures 4 and 5. The **`pure model`** methods `scaleTime` and `getDimensions` are introduced in `Vector2d` and used in the duration specification rewritten for `Vector2d` and inherited by `Vector3d`.

This approach introduces a practical problem related to 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. Clearly, evaluating these methods at execution time is a process that takes time, and might affect the overall timing analysis of the program. A practical and precise solution remains an open topic for research and future work. For instance, partial evaluation techniques can be used to eliminate the need for many time-demanding calculations at runtime, by doing some calculations during the program's compilation. Another possibility is to postpone many calculations to checking time, by putting the information needed to do such calculations into the execution traces.

3. DISCUSSION

In this section, we introduce one more example inspired by one of the SCJ examples. Figure 6 declares a type `ListHandler` (itself a subtype of `PeriodicEventHandler`) that inverts a list when a specific event occurs. In Figure 7, we introduce another type, `SortedListHandler`, which is a subtype of `ListHandler`. When the event occurs a `SortedListHandler` object sorts the list instead of inverting it. The specifications for both types show how the suggested methodology can be used to specify timing constraints. For the subtype `SortedListHandler`, since it is a behavioral subtype of its super type `ListHandler`, specifications from both types must be satisfied. However, following our proposed approach, the duration clause is only specified in the super type, and only the pure model method, `handleAsyncEventTime`, used by the duration clause, is overridden in the subtype in order for the duration clause to be satisfied.

4. RELATED WORK

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

Our solution for the subtyping problem was based on the work of Parkinson and his coauthors [28, 27]. In his work, Parkinson

```
public class ListHandler extends
     PeriodicEventHandler {
    //@ public model instance int count;
    //@ public initially count == 0;
    //@ public invariant count >= 0;

    protected /*@ spec_public @*/ int count_;
    //@ in count;
    //@ public represents count = count_;
    protected /*@ spec_public @*/ List list;

    // ...

    /*@ public normal_behavior
      @   requires count >= 0 && list != null;
      @   assignable count, theMission;
      @   ensures count == \old(count) - 1;
      @   ensures list.size()
      @             == \old(list.size());
      @   duration  handleAsyncEventTime();
          @*/
    public void handleAsyncEvent() {
       // ...
       list.invert();
       if (--count_ == 0) {
          getCurrentMission().
                requestSequenceTermination();
       }
       // ...
    }
    /*@
    public pure model long handleAsyncEventTime()
        {
        return (long)
            (0.5 * ITimeConstants.ObjectSwapTime);
        }
    @*/
    // ...
}
```

Figure 6: Specifications for `ListHandler`.

developed a formal semantics for a subset of Java, and introduced the concept of abstract predicate families to modular reasoning of specifications to address the modularity of reasoning in the presence of inheritance and subtyping. 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. Parkinson and his coauthors do not extend their work to timing specifications, which is our application of the concept in the present paper.

The design and implementation of the **`duration`** clause in SafeJML (and in JML) is based on the work of Krone *et al.* [15, 16]. Much of SafeJML's design is built to accommodate the RapiTime tool [1, 14], a hybrid analysis tool used to perform hybrid WCET analysis for C programs.

Formal verification of timing specifications is also introduced in Hehner's work [13]. Hehner used refinement calculus to formalize the verification of timing specifications, but he does not discuss OO issues such as subtyping.

Many researchers have worked on the problem of WCET analysis. One early work in this area is Shaw's book [33]. 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.

```
public class SortedListHandler extends
    ListHandler {
   //@ public model instance int n;
   //@ public initially n == 0;
   //@ public invariant n >= 0;
   //@ protected represents n = list.size();

  /*@ also
    @  public normal_behavior
    @    requires n >= 1;
    @    assignable count, theMission;
    @    ensures list.isSorted();
    @*/
   public void handleAsyncEvent() {
      // ...
      list.sort();
      if (--count_ == 0) {
         getCurrentMission().
               requestSequenceTermination();
      }
      // ...
   }
   /*@
   public pure model long handleAsyncEventTime()
       {
       return ITimeConstants.ObjectSwapTime
               * n * (long)(Math.log(n));
       }
   @*/
   // ...
}
```

Figure 7: Specifications for `SortedListHandler`, a subtype of `ListHandler`.

Schoeberl and Pedersen [32] 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 the type at compile time.

5. CONCLUSION

In this work, we have described a solution to the problem of specification and checking of timing constraints in SCJ programs that use subtyping. Our solution is embodied in SafeJML, an extension to JML for specification of SCJ programs. We showed how the duration clause can be used to check SCJ programs that use subtyping by writing specifications that make use of model methods (Parkinson's abstract predicates).

Future work includes a full implementation and evaluation of the proposed solution based on real life examples. We also plan to investigate and introduce more issues regarding the use of other techniques and features of object-oriented languages in real-time systems.

Acknowledgment

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

APPENDIX

A. INSTALLATION INSTRUCTIONS

We provide a wiki page for SafeJML at `http://tinyurl.com/28zllux`. The page contains documentation on how to build and test SafeJML.

B. REFERENCES

[1] 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.

[2] 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*, 7(3):212–232, June 2005.

[3] 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*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer-Verlag, 2006.

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

[5] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis, Portland, OR*, pages 24–27, New Mexico State University, May 2003. Jonathan Cook.

[6] 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)*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer-Verlag, 2001.

[7] J. Gustafsson. Worst case execution time analysis of object-oriented programs. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 0:0071, 2002.

[8] 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, pages 155–163, New York, NY, USA, 2010. ACM.

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

[10] I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001.

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

[12] 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*, 91(7):1038–1054, July 2003.

[13] E. C. R. Hehner. Formalization of time and space. *Formal Aspects of Computing*, 10:290–306, 1998.

[14] 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*, pages 67–70, June 2004.

[15] 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)*, pages 60–67, 2001.

[16] J. Krone, W. F. Ogden, and M. Sitaraman. Profiles: A compositional mechanism for performance specification. Technical Report 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.

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

[18] G. T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 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.

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

[20] 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*, 31(3):1–38, Mar. 2006.

[21] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, Cambridge, UK, 2000.

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

[23] 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.

[24] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995.

[25] K. R. M. Leino. Recursive object types in a logic of object-oriented programs. *Nordic Journal of Computing*, 5(4):330–360, Winter 1998.

[26] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[27] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In P. Wadler, editor, *ACM Symposium on Principles of Programming Languages*, pages 75–86, New York, NY, Jan. 2008. ACM.

[28] M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, Nov. 2005. The author's Ph.D. dissertation.

[29] 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, pages 69–82, New York, NY, USA, 2010. ACM.

[30] 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*, pages 95–101, New York, NY, USA, 2010. ACM.

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

[32] 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*, pages 202–211, New York, NY, USA, 2006. ACM.

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

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

[35] J. M. Wing. Writing Larch interface language specifications. *ACM Trans. Prog. Lang. Syst.*, 9(1):1–24, Jan. 1987.