The Design of SafeJML, a Specification Language for SCJ with Support for WCET Specification

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

Faraz Hussain
University of Central Florida,
Orlando, FL, USA
fhussain@eecs.ucf.edu

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

ABSTRACT

Safety-Critical Java (SCJ) is a dialect of Java that allows programmers to implement safety-critical systems, such as software to control airplanes, medical devices, and nuclear power plants. SafeJML extends the Java Modeling Language (JML) to allow specification and checking of both functional and timing constraints for SCJ programs. When our design of the SafeJML is implemented, it will help check the correctness of detailed designs, including timing for real-time systems written in SCJ.

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

Safety-critical systems are systems in which an incorrect response or an incorrectly timed response can result in significant loss, including loss of life. Safety-critical systems are usually also real-time systems. They must also be extremely reliable and safe, as they control critical systems like airplanes and nuclear reactors. Moreover, response time is crucial in these systems, as they interact with a hardware interface [3].

Software reliability is a major issue for safety-critical systems. Many life-threatening incidents 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 [28]. Many other examples can be found in literature and news [29]. Thus safety-critical systems require a rigorous validation and certification process. For example, in the United States, the Federal Aviation Administration requires certification of aircraft software using the DO-178B standard [33].

One benefit of using SCJ is that, as a dialect of Java, it promises cost savings in development and maintenance compared to programs written in lower-level languages, such as C or assembler. Such cost savings could greatly aid real-time and safety-critical embedded systems, given their intrinsically higher overall costs. In particular, using SCJ gives the potential for better tool support, such as integrated development environments, and will allow more seamless development of embedded systems that are tightly integrated with (non-critical) back-end systems.

The Java Modeling Language (JML) [2, 5, 23, 22, 26] is a behavioral interface specification language [38] that boasts many state-of-the-art features for functional specification.

JML's design currently has minimal support for specification of space and timing constraints, based on the work of Krone *et al.* [20]. In this paper we focus on timing constraints, which in JML appear 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 "JVM cycles" [26]. However, the `duration` clause has never been implemented in the JML tools or used in actual case studies. Further, it is not clear that measuring speed in JVM cycles is appropriate for real-time systems.

Our 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. We describe both our rationale for the language design and our initial design of an implementation. To our knowledge, this is the first publicly released and documented real-time extension to JML. The main contribution of this work is the design of SafeJML.

2. AN OVERVIEW OF SAFEJML

SafeJML is designed to allow SCJ users to specify both functional and timing behavior on a per-method basis.

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 (rare) runs of a task that exceed their time budget. 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 may be contrary to the way that real-time programmers typically work, we note that such SafeJML specifications can be added to methods in layers

JTRES'10 August 19-21, 2010 Prague, Czech Republic

(first to the methods called directly from a task, 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.

In the remainder of this section we first give an overview of SCJ and the implementation of SCJ that we use — the oSCJ [7] virtual machine. While there is also a reference implementation of SCJ available, the oSCJ implementation is the most mature implementation, and, because it translates SCJ programs into C code, allows us to work with low level WCET analysis tools.

We also introduce RapiTime [19] as an example of how SafeJML can integrate with timing analysis tools. Our design of SafeJML is intended to be usable with other timing analysis tools, for example AbsInt's aiT static analysis tool [9, 15, 14]. However, we chose RapiTime because it enables us to use information obtained from runs of a program (dynamic analysis) to check if any timing specifications were violated. Furthermore, the annotation information that RapiTime can process is a superset of that understood by aiT. Thus by targeting RapiTime, we have a good start on making our tool work for both RapiTime and aiT.

2.1 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 [37]. 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 [33] and other safety-critical standards. JSR-302 is near completion now [17].

Vitek's group at Purdue has produced oSCJ [7], an open-source SCJ implementation based on OVM [32]. (Currently oSCJ implements all of Level 0 of SCJ [37].) 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. This process allows SafeJML to use WCET tools that are targeted at C program object files (as opposed to Java bytecodes). 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. (SafeJML is designed to specify such C code by specifying the Java Native Interface (JNI) declarations used to interface the Java code to the C code.)

2.2 Timing Analysis with RapiTime

Standard WCET analysis tools such as aiT use static program analysis techniques. This approach requires having a precise timing model for the target processor. By contrast, the RapiTime tool [19], uses a hybrid timing analysis using runtime measurements together with static program analysis [1]. This approach works on a large variety of processors, perhaps because it does not involved detailed modeling of each processor's architecture.

For these reasons both the oSCJ team at Purdue and our team at the University of Central Florida have decided to use RapiTime as the main analysis tool. However, we intend the design of SafeJML to also allow the use of other tools, such as aiT. As mentioned above, another reason for our decision is that RapiTime seems to accept more information from user annotations than aiT, and so a design of SafeJML that targets RapiTime should also allow the use of aiT.

In order for RapiTime to do timing analysis, it first derives a static model by parsing the source code to build a control flow graph. Then the user runs various test inputs on the target machine, while RapiTime records execution times for the program's basic blocks. RapiTime then uses the longest recorded time for each basic block and information from static analysis or from user annotations about the maximum number of times control may pass through various basic blocks (such as loop bodies) to form a timing model. RapiTime (under-) approximates the worst case execution times of each method by multiplying the maximum number of executions for each basic block and the maximum recorded execution time for that block. The model RapiTime constructs can also provide a path-sensitive calculation of the worst case execution time for a program, by using the program's call graph.

Note, however, that RapiTime's technique for approximating the worst case execution time of a basic block requires the user to provide sample inputs (test cases) that exercise the worst case execution time for that block. To get close to the true worst case for a method, independent of the program in which the method is used, the user would also have to select inputs that produce all processor states that could cause worst case times (e.g., by requiring caches to be flushed).[1] In any case such inputs must provide complete code coverage. The possibility that the user might not know how to provide such inputs is the main disadvantage of this approach, which could lead to under-approximation of the worst case execution time. However, this disadvantage can be mitigated to some extent by providing more sample inputs.

RapiTime can be considered path-sensitive; although it needs the code to be executed in order to perform analysis, it collects timing information for each execution path, and then it uses the program's call graph and flow graph to approximate the WCET for each path.

2.3 A First SafeJML Example

In this subsection we give a first example of SafeJML. This example is taken from MiniCDj, a SCJ rewrite of the CDx benchmark suite [18]. The CDx benchmark suite is an open-source family of collision detection benchmarks that can measure performance of various real time platforms. The example is a method call that is responsible for computing the motions and the current positions of aircrafts; it then stores the positions of the aircrafts in a state table.

The package uses the method call `Benchmarker.set()` and `Benchmarker.done()` (lines 2 and 26, respectively) as an anchor for RapiTime to measure the time spent between the two calls. RapiTime then uses static analysis techniques to calculate the WCET for different program paths.

To improve RapiTime's timing analysis, RapiTime uses various annotations to restrict loop bounds, and restrict execution times, and prune infeasible paths. SafeJML has annotation comments (which look like `/*@ ... @*/` or lines starting with `//@`) that translate into the corresponding RapiTime annotations. For example, in Figure 1, line 9 specifies the maximum number of times the loop may iterate. Line 14 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. Lines 14 and 20 specify that the two paths in which these lines are included are mutually exclusive.

A key requirement in the implementation of SafeJML is to translate these annotations to the corresponding RapiTime annotations.

[1] It is not clear how well the user can control the generation of different processor states in the case of multiprocessors, where cache and memory contention may be very difficult to predict.

```
1  public List createMotions() {
2    Benchmarker.set(6);
3    final List ret = new LinkedList();
4
5    Aircraft craft;
6
7    for (int i = 0, pos = 0;
8         i < currentFrame.planeCnt; i++) {
9      /*@ maximum_loop_iterations
           MAX_PLANES_PER_FRAME; @*/
10     // perform some calculations...
11     if (old_pos == null) {
12       // new aircraft; calculate more...
13       //@ path new_aircraft;
14       /*@ local_worst_case
             MAX_NEW_PLANES_PER_FRAME; @*/
15       ...
16     } else {
17       // old aircraft; other calculations...
18       /*@ path old_aircraft
               \exclude new_aircraft; @*/
19       ...
20       }
21
22     ret.add(m);
23     }
24   }
25   Benchmarker.done(6);
26   return ret;
27 }
28
```

Figure 1: The `createMotions` method from the collision detection benchmark, with SafeJML annotations added.

3. SAFEJML SYNTAX AND SEMANTICS

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

3.1 Statement Annotations

SafeJML, like JML [26], 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 the RapiTime tool (or other, similar, WCET tools). We discuss the details of these statements below.

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

3.1.1 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 loop-max-iter-stmt is used to specify the maximum number of iterations the loop can have. This annotation has one argument, a *constant-expression*, which is a (side-effect free) Java expression that can be resolved at compile time, and must be of type **`int`** or **`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 following is the syntax of the SafeJML annotated loop statements.

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

The example in Figure 2 includes a *loop-max-iter-stmt*. In this example, the constant `MAX_ARR_SIZE` is used with prior knowledge from the programmer to bound the loop with an upper limit.

```
//@ public abstract model int MAX_ARR_SIZE = 100;

public abstract class SumArrayLoop {
   public static long sumArray(int [] a) {
      long sum = 0;
      int i = a.length;
      while (--i >= 0){
         /*@ maximum_loop_iterations
              MAX_ARR_SIZE; @*/
         sum += a[i];
      }
      return sum;
   }
}
```

Figure 2: SafeJML maximum loop iterations statement annotation.

3.1.2 Local Worst Case Statements

SafeJML has a new annotation construct for blocks of code nested inside conditional statements, the *local-worst-case-stmt*. 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 also within a conditional statement (such as an **`if`** or **`switch`** statement).

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

The *local-worst-case-stmt* has one argument, a *constant-expression*. This argument denotes the maximum number of times that all paths on which the *local-worst-case-stmt* appears may execute during an instance of the enclosing loop. Figure 3 gives an example.

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

Figure 3: SafeJML local worst case statement annotations.

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.

3.1.3 Path Annotations

From RapiTime, SafeJML takes the concept of specification of paths, 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. To declare a new path group, the new name has to appear in the **\in** or **\exclude** clause of the "path" statement. If neither **\in** or **\exclude** clauses are used, then the path group for that path is just the singleton group, whose name is the same as the same path name. The main purpose for introducing path groups is to be able to optimize the WCET analysis by reducing the number of paths to be analyzed, and improve the WCET estimate by explicitly describing infeasible paths.

The path name **\default** is used to represent the default execution path of the code, which will include the WCET path after analysis, so the annotation statement

```
path myPath \in \default;
```

will not have any affect on the path group of `myPath`, but using

```
path myOtherPath \exclude \default;
```

will cause the analysis to exclude the path named `myOtherPath` from the analysis. Excluding a 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.

The following is the syntax for the *path-anchor-stmt*, which is used to describe path names and path groups.

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

Figure 4 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.

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.

Figure 5 shows a small example on how path and path group annotations can be used in switch statements.

3.2 Contract Clauses

SafeJML, like JML, specifies methods using contracts ("specification cases") [26]. 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 following, which we discuss below.

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

```
1  void search( int limit ) {
2     /* ... */
3     if(count != 0){
4        //@ path char_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 4: SafeJML path annotations used with if statements.

```
void method1 ( int arg1, int arg2 )
{
   if (arg1 == 0){
      //@ path path_a;
      /* ... */
   }
   switch(arg1) {
   case 0:
      //@ path path_b \in path_a;
      /* ... */
      break;
   default:
      //@ path path_c \exclude path_a;
      /* ... */
      break;
   }
}
```

Figure 5: SafeJML path annotations in a switch statement.

3.2.1 Context-Dependent Annotations

A static analysis method is *context-dependent* if it performs different analyses for method calls depending on the place in the code where the method is called. Context dependency can aid the precision of WCET analysis if different calls of a method will have greatly differing timing behavior.

Following RapiTime, SafeJML allows annotations on method specifications to specify when RapiTime (or some other tool) should track the context of method calls. This is the *splits-wcet-clause*. The clause gives a predicate that describes if a context-dependent analysis should be used.

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

Figure 6 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 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 passed parameter, then this clause will provide the 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.

```
/*@ splits_wcet true; @*/
void splitter(int arg1) {
   /* ... */
}

/*@ splits_wcet arg1==3; @*/
void splitIf3(int arg1) {
   /* ... */
}

void method1() {
   splitter(1); // context-dependent analysis
   splitter(2); // context-dependent analysis
   splitter(0); // context-dependent analysis
   splitIf3(1); // unified analysis
   splitIf3(2); // unified analysis
   splitIf3(3); // context-dependent analysis
}
```

Figure 6: SafeJML example using the `splits_wcet` clause.

3.2.2 Duration Annotations for Methods

SafeJML, like JML itself [26] has a **`duration`** clause, that is intended for specifying the worst case execution time of a method (or block of code).

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

The semantics of JML's duration clause is based on the work of Krone *et al.* [20, 21]. It specifies the maximum execution time needed for a method call (for a given precondition, since it forms part of a specification case [26, 22]). Unfortunately, JML's duration clause measures execution time in "JVM cycles," which does not make it easy to match against software requirements stated in absolute time units (such as seconds). The fiction of a standard JVM cycle time also does not match reality, in which Java bytecodes may take different execution times. Even if there were a table of how many "JVM cycles" each bytecode would take to execute, this idea would not be a good match for compiler-based virtual machines, such as the OVM, which translate SCJ code into C code.

Thus, in SafeJML, we change the semantics of the duration clause and measure time in nanoseconds (as RapiTime also uses nanoseconds as units). We assume that the built-in SafeJML package named `org.jmlspecs.lang` contains definitions of constants such as `MILLISEC`, `SEC`, etc. to allow expression of timing constraints in units that are more convenient for the specifier.

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 more than one specification case holds, then each of the corresponding specification cases must have all their clauses satisfied by the method's execution (this is the reason the keyword is "also") [22]. 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 7 shows how the **`duration`** clause is used 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

```
/** 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) {
  Benchmarker.set(7);
  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;
  Benchmarker.done(7);
}
```

Figure 7: SafeJML method specification example. The method's specification has 3 specification cases, which are separated by `also`.

each specification case contains a requires clause and a duration clause. The duration clause will be used by SafeJML during the WCET analysis for the method. The SafeJML tool will check that the WCET for this method is no larger than the specified time for each case, given that the method's position argument satisfies the corresponding precondition. The tool also uses these expressions for WCET 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 WCET of that call. (If at a call site one can only prove that some precondition holds, but not which one, then one can use the maximum of all the duration clauses during verification.)

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 [26, 35] contains the following syntax.

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**

The meaning of a statement **`refining`** S C is that the code C has to satisfy the specification S. For example, the following code

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

says that the call to `m` may take at most 3 milliseconds.

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 could be useful when calling methods that do not have SafeJML specifications.

3.4 Subtype Polymorphism Annotations

A major intellectual challenge in our work is the object-oriented (OO) nature of SCJ. That is, the dynamic dispatch mechanism of Java, which is also present in SCJ, permits *subtype polymorphism*: the ability to make the same method call on objects of all subtypes of a given type [4]. In many works on real-time systems, authors suggest disallowing subtype polymorphism [10]. In addition to such healthy conservatism, one reason for avoiding subtype polymorphism is that applying behavioral subtyping to timing constraints poses a practical problem. This problem is that instances of a proper subtype of an OO type often contain more information than instances of its supertypes. So if the supertype's method has a very tight timing constraint (given by its duration clause), that specification may not allow enough time to execute a subtype's overriding method. The alternative, underspecification of the supertype's method, would, by definition, not permit tight timing constraints.

In sum, the problem is how to specify (supertype) methods in a way that allows some subtypes to use more time, while retaining modular verification and precision (tightness of a timing analysis). Since this problem is both a problem of specification and of verification, we treat specification and verification separately below.

For specification, we follow recent specification ideas [6, 31, 30] by including in SafeJML features that allow users to write specifications with meanings that vary with the dynamic type of the method's receiver. In SafeJML this is accomplished by writing specifications using a (JML) feature called "model methods." These are side-effect free (pure) methods that are written to be used in specifications. As shown by Parkinson's work with the equivalent of model methods,[2] the ability to write different implementations of model methods in subtypes allows the meaning of a model method to vary depending on the dynamic type of the receiver; it can thus have different meanings in each subtype. For example, one could specify a method `m` with a duration clause **`duration`** `mTime();`, where `mTime` is a pure model method. And then a call such as `o.m()` would be known to have a WCET of `o.mTime()`, where the exact value of `o.mTime()` may vary with the runtime type of the receiver, `o`.

Parkinson's work gives sound rules for reasoning about specifications that are written using such model methods. In essence, during verification one must either have a type-independent specification of the model method's meaning or one must perform a case analysis on the runtime type of the receiver. In the first approach, we can take advantage of a property of SafeJML that, as in JML, the specification of model method must be obeyed in all subtypes [22]. Thus, if the static type of the receiver (i.e., `o`'s type) has a specification for the model method (`mTime`) that is sufficiently exact, then one can use that specification to compute the WCET (i.e., to bound the value of the call `o.mTime()` and hence to bound the time of `o.m()`).[3]

The second approach is needed when the model method (`mTime`) does not have a sufficiently exact specification (to draw the desired conclusions). In this case, during verification one would need to be able to prove some tight bounds on the runtime type of the receiver (`o`). That knowledge would allow the verification to use the definitions of the model methods that are given for the possible receiver types.

To facilitate this second kind of reasoning we advocate using SafeJML's **`assume`** statements together with a built-in predicate named `SafeJML.type_bound`. An **`assume`** statement gives a predicate that is believed to be true at runtime, and can be checked by a runtime assertion checker. `SafeJML.type_bound(`S,E,T`)` is true just when the value of the expression E has a dynamic type that is between types S and T (i.e., a supertype of S and a subtype of T). Assumptions written using `SafeJML.type_bound` can thus restrict the cases that need to be considered during verification.

Figure 8 gives an example of the usage of the subtype polymorphism annotations. In this example, `v_pre` is declared to have type `Vector`. Thus the calls on lines 28 and 32 to the method `getIntersection()` method will be dynamically dispatched. Line 27 helps the tool to determine a subset of methods that the call can be dispatched to by narrowing the methods to be considered. In this case the receiver has dynamic type `Vector2d`. Line 31 serves the same purpose.

3.5 Error reporting

For statement annotations like the *max-loop-iter-stmt* and the *local-worst-case-stmt*, SafeJML reports the violation to the user by throwing an error (a subtype of `JMLAssertionError`) as soon as it is detected. However, for other constructs (like the *duration-clause*) an error can only be flagged at program termination because their semantics under SafeJML depends on information received from a tool like RapiTime.

4. SAFEJML IMPLEMENTATION

Our implementation of SafeJML is built on the JAJML compiler [13]. JAJML is built using the JastAdd tool for developing extensible compilers [8]. JAJML also builds on JastAddJ, the Java compiler implemented by the makers of JastAdd. JAJML is open source, as is SafeJML. SafeJML's code is available using subversion from `http://refine.eecs.ucf.edu/svn/scjml`.

SafeJML extends JML by introducing the constructs described in the previous section. However, there are restrictions as to where in a piece of code, these new constructs will be recognized by the parser. For example, the **`maximum_loop_iterations`** annotation can only be used as the first statement inside a loop.

In the implementation we have used JastAdd's synthesized and inherited attributes, which provide convenient mechanisms for propagating type information around the abstract syntax tree of an SCJ program. So far, we have implemented the typechecking for loop maximum iteration statements and local worst case statements. The restrictions on the use of other SafeJML annotations can be implemented in a similar way.

One limitation in our implementation is that duration annotations (Section 3.2.2) are not implemented completely yet. This is because JML's specification statements [25, Section 14.6] are currently not implemented in JAJML.

The most challenging part of the implementation, translating SafeJML annotations into the annotations needed by RapiTime, is future work.

5. SAFEJML EVALUATION

To gain more confidence with the proposed language design, we

[2] Parkinson calls them "abstract predicate families."

[3] Reasoning about OO programs using the specifications of each receiver's static type is called "supertype abstraction" [27, 22, 24].

```
1  class Vector {
2     public abstract Point getIntersection(Vector
          v);
3  }
4  class Vector1d extends Vector {
5     public Point getIntersection(Vector v); { /*
          ... */ }
6  }
7  class Vector2d extends Vector {
8     public Point getIntersection(Vector v); { /*
          ... */ }
9  }
10 class Vector3d extends Vector {
11    public Point getIntersection(Vector v); { /*
          ... */ }
12 }
13 class client {
14    Vector v_pre;
15    Vector v_post;
16    Vector[] data = new Vector[5];
17    public void client() {
18       v_pre = new Vector2d();
19       v_post = new Vector3d();
20       for (int i=0;i<5;i++) {
21          if (i%2 != 0) { data[i]=new Vector2d();
                }
22          else { data[i] = new Vector3d(); }
23       }
24    }
25    public List<Point> getIntersections() {
26       List<Point> points = new
             LinkedList<Point>();
27       /*@ assume SafeJML.type_bound(Vector2d,
             v_pre, Vector2d); @*/
28       points.add(v_pre.getIntersection(v_post));
29       for (int i=0;i<4;i++)
30       {
31          /*@ assume SafeJML.type_bound(Vector2d,
                data[i], Vector3d); @*/
32          points.add(data[i].getIntersection(
33                                      data[i+1]));
34       }
35       return points;
36    }
37    public static void main(String[] args) {
38       client c = new client();
39       System.out.println(
40             c.getIntersections().toString());
41    }
42 }
```

Figure 8: SafeJML example that uses assumed type bounds to limit the range of types that must be considered during analysis.

picked several examples from MiniCDj and specified them. A first example is shown in Figure 1. This example shows clearly the ability of the language to reduce the complexity of the analysis.

Another example is shown in Figure 9. Line 6 specifies the maximum number of motions the system can encounter in a single frame. Since we can argue that the system can only have as many motions as the number of planes the system can encounter, it is safe to specify the loop this way. Since the inner loop will execute one time less than the outer loop, it is specified as in line 9. The last specification statement, on line 13, can be used in a real life situation, if the assumption is made such that the system is good enough if it can detect at least one collision. However, in a simulated environment, this assumption does not hold, hence, it is safer to omit this specification statement, and leave the **if** statement unspecified.

```
1  public int determineCollisions(final List
       motions, List ret) {
2    Benchmarker.set(5);
3    int _ret = 0;
4    Motion[] _motions = (Motion[])
         motions.toArray(new
         Motion[motions.size()]);
5    /*@ assume MAX_PLANES_PER_FRAME <
         _motions.length; @*/
6    for (int i = 0; i < _motions.length - 1; i++) {
7      /*@ maximum_loop_iterations
           MAX_PLANES_PER_FRAME; @*/
8      final Motion one = _motions[i];
9      for (int j = i + 1; j < _motions.length - 1;
           j++) {
10       /*@ maximum_loop_iterations
             MAX_PLANES_PER_FRAME - 1; @*/
11       final Motion two = _motions[j];
12       final Vector3d vec =
             one.findIntersection(two);
13       if (vec != null) {
14         ret.add(new Collision(one.getAircraft(),
               two.getAircraft(),vec));
15        _ret++;
16       }
```

Figure 9: The `determineCollisions` method from the collision detection benchmark, with SafeJML annotations added.

Another example is shown in Figure 9. Line 7 specifies the maximum number of motions the system can encounter in a single frame. Since we can argue that the system can only have as many motions as the number of planes the system can encounter, it is safe to specify the loop this way. This assumption is recorded in line 5. Since the inner loop will execute at most one time less than the outer loop, it is specified as in line 10.

Note that since we require a constant expression for the bound in a **`maximum_loop_literations`** annotation, the bound given in line 10 is quite conservative.

For the block in lines 14-15, if one assumed that the system can detect at most one collision (i.e., assuming others would have been detected earlier and avoided), then one could use the annotation **`local_worst_case`** `1;` in this block. However, in a simulated environment, this assumption does not hold, so we omit this annotation.

6. RELATED WORK

We know of no other design of a specification language for SCJ.

A well-known example of WCET static analysis is Shaw's book [36]. Shaw discusses measurement and analysis methods that use

path expressions for a precise (tight) analysis. However, Shaw's is a whole program analysis, which is not designed for modular verification, and it does not consider subtype polymorphism for OO languages.

Schoeberl and Pedersen [34] describe a precise WCET 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. 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 more refined case analysis to handle subtype polymorphism, as described in Section 3.4.

Hehner [16] formalized verification of timing constraints by introducing a ghost (specification-only) variable t to represent the current time. He use standard axiomatic reasoning techniques (in a refinement calculus style) to specify timing constraints in postconditions and to do static verification. While it is modular, since it is based on specifications, and while it describes how to automate verification, Hehner's work does not provide an interface to verification tools such as RapiTime or aiT. It also does not treat OO language features.

Both JML and our work build on the work of Krone *et al.* [20, 21]. They use specifications augmented with a **`duration`** clause that can state timing constraints. These clauses can depend on parameters and on the post-state of a method (which is useful for underspecified methods). Their work supports modular verification of timing constraints using specifications, however, it does not deal with OO features. JML adopts their design for the **`duration`** clause, which our work takes as its starting point.

Many ideas in SafeJML are taken from RapiTime, a tool that uses a hybrid WCET analysis using runtime measurements together with static program analysis [1, 19]. 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 paper). Such specification facilities may be quite useful for safety-critical systems.

Gustafsson *et al.* [12] suggest using abstract execution to aid WCET analysis. They introduced their idea first in [11]. Abstract Execution is a form of symbolic execution which is based on abstract interpretation. 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 is not modular.

7. CONCLUSION

In this work, we have described the design of SafeJML, an extension to the Java Modeling Language (JML) for specification of safety critical Java programs. SafeJML's design as a JML extension supports the modular specification of both functional and timing constraints for SCJ programs. We are implementing SafeJML as an extension to JAJML, an open-source extensible JML compiler.

Future work includes integration of the nascent SafeJML tool with RapiTime via the oSCJ virtual machine, and evaluating the utility of the language design with feedback from more extensive case studies. Besides evaluating the design of SafeJML more fully, we also expect to evaluate different specification and verification methodologies. In particular, we would like to understand better the tradeoffs in using underspecification vs. case analysis to specify subtype polymorphism in safety-critical and 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] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longmain, 3 edition, 2001.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.

[5] 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.

[6] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. In P. Wadler, editor, *ACM Symposium on Principles of Programming Languages*, pages 87–99, New York, NY, Jan. 2008. ACM.

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

[8] T. Ekman and G. Hedin. The JastAdd system — modular extensible compiler construction. *Sci. Comput. Programming*, 69(1-3):14–26, 2007.

[9] 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.

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

[11] 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*, 0:257, 1997.

[12] 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*, 0:57–66, 2006.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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*, pages 41–50, New York, NY, USA, 2009. ACM.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, and J. Kiniry. Jml reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, Apr. 2003.

[26] 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.

[27] 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.

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

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

[30] 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.

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

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

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

[34] 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.

[35] 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*, pages 351–367, New York, NY, Oct. 2007. ACM.

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

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

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