Provably Correct Loops Bounds for Realtime Java Programs

James J. Hunt and Fridtjof B. Siebert
aicas GmbH
Haid-und-Neu-StraÃe 18
D-76139 Karlsruhe, Germany
jjh|siebert@aicas.com

Peter H. Schmitt and Isabel Tonin
University of Karlsruhe
Am Fasanengarten 5
D-76131 Karlsruhe, Germany
pschmitt|tonin@ira.uka.de

ABSTRACT

Determining concrete bounds for loops is one of the more vexing problems of resource analysis of realtime programs. Current mechanisms are limited in scope and require considerable user input that can not be verified. The authors present a methodology for providing more general loop bounds where the correctness can be demonstrated with formal techniques. The methodology combines data flow analysis and deductive formal verification to attain this goal.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Algorithms, Performance, Reliability, Theory, Verification

Keywords

realtime Java, worst case execution time, worst case memory usage, formal analysis, data flow, deductive verification

1. INTRODUCTION

Analysis of runtime performance and memory use is an essential part of ensuring that a realtime programs runs correctly under all operating circumstances. These systems depend on the ability to response in realtime to properly fulfill their task: they must always be able to complete some task within a predefined time interval whenever a predefined event occurs. Likewise, they must not use more memory that the system can provide. Testing can be used to see if such time and space bounds are usually met, but can not prove that the task will always finish within its time and space bounds. Only formal analysis can provide this level of assurance.

Both worst case execution time analysis (WCETA) and worst case memory usage analysis (WCMUA) must examine all possible execution paths of a subprogram. WCETA must determine which path would take the most time to execute; whereas, WCMUA must determine what the maximum amount of allocated memory is. In both cases, the analysis is dependent on the complexity of a given subprogram and under what conditions it terminates. This, in turn, is strongly related to the upper bounds of loops iterations and recursion depths; however, establishing these bounds is non trivial. The state of the art is to require the program developer to specify them for each application in one form or another.

The authors present a new method of bounding loops that uses advances in data flow analysis and deductive formal verification for Java programs to enable one to provide parametrized, provably correct bounds for loops. The technique has the advantage that user input can be verified for correctness and is also reusable. Since, at least tail recursion is equivalent to looping and more general recursion is usually not applicable in realtime programs, this loop bounds analysis is sufficient for the vast majority of realtime programs.

2. STATE OF THE ART

Much more work has gone into execution time analysis, in particular worst case execution time analysis than memory usage analysis. Where as critical system can avoid other resource usage problems by allocating all resources in an initialization phase and limiting the program to these resources, such techniques do not work with program execution. As a result, much more has been published about WCETA than WCMUA.

WCETA has been a part of realtime system design for quite a long time, but even here literature is relatively young. Before the advent of caching and pipelines in embedded processors, it was common practice to calculate WCET by hand using a table of instruction execution times given in clock cycles. Static analysis papers started to appear at the end of the 80's, e.g. the work of Messrs. Puschner and Koza[25].

Most of the activity in the field is centered in Europe, where there are still a handful of groups active including the TU Vienna in Austria, Saarbrücken in Germany, INRIA/IRISA in France, the University of York in the UK, and the ARTES network of the Uppsala University, the Mälardalen University, Chalmers University of Technology, and Lund Institute of Technology, all in Sweden. Unfortunately, being research projects, very few tools are really usable beyond the laboratories in which they were designed. Information about them can only be inferred from the pa-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
The 4th Workshop on Java Technologies for Real-time and Embedded Systems - JTRES '2006 Paris, France
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

pers written about them. A contributing factor to the dearth of information is the hope of marketing the tools and the consequent desire to limit free access.

There are many interesting aspects of WCETA, but the only one germane here is the handling of loop and recursion bounds. Much of the work has concentrated on machine dependent aspects of WCETA, such as pipeline, cache, branch prediction, and instruction dispatch analysis. Much less has been done with machine independent aspect of analysis, such as loop bounds and dispatch set analysis.

2.1 CALC_WCET_167

CALC_WCET_167 [18] was developed in the Real-Time Systems Research Group at the Vienna University of Technology. This tool is designed to provide WCET analysis on the Siemens C167CR architecture (hence the name). It is tightly linked with a special version of GCC (the GNU C Compiler) for the C167 which uses WCETC [19], a derivative of the C programming language, as its input language. WCETC includes support for specifying the loop bound on the loop headers (do..while, while, and for loops) and the maximum execution count of certain paths in the flow graph. It forbids the use of function pointers, recursive function calls, goto, setjmp(), longjmp(), signal() and exit().

2.2 Cinderella 3.0

One of the first tools to be able to model advanced processor architectures, Cinderella has some interesting features, as it was designed to be easily retargeted to other architectures. The author provides architecture models for the Intel i960 and the Motorola M68000. Cinderella performs its analysis on the executable files in COFF format in both architectures, but it requires the user to specify via the provided GUI the minimum and maximum loop iteration bounds.

2.3 Heptane

Heptane[1][9, 10] was developed by IRISA as part of the Hades project[2]. Heptane is a fairly complete tool that models a very recent processor: the Intel Pentium. Heptane is built on top of Hexane, a preprocessor that reads the C source file and outputs the syntactic tree of the program under analysis, and Salto, which provides the hardware dependent information. Maximum loop iteration is provided by the user in the form of source code annotations. Some restrictions on the source code are that the use of function pointers is forbidden and features that may cause unstructured control flow graphs (i.e. control flow graphs that are not compatible with the syntax tree) are not allowed including the use of the goto construct and the inclusion of assembly code that contains branching instructions.

2.4 aiT

aiT is a fairly complete tool produced by AbsInt, in close cooperation with the Universität des Saarlandes in Saarbrücken. It takes as input the executable file in ELF or COFF formats. Dynamic data structures, and setjmp(), longjmp() calls can not be analyzed. aiT tries to determine the number of loop iterations by loop bound analysis, but succeeds in doing so for simple loops only. Bounds for the iteration numbers of the remaining loops must be provided as user annotations. Loop bound analysis relies on a combination of value analysis and pattern matching, which looks for typical loop patterns. aiT assumes that the generated code is well behaved. In general, these loop patterns depend on the code generator and/or compiler used to generate the code that is being analyzed. There are special aiT versions adapted to various generators and compilers.

[1]Hades Embedded Processor Timing ANalyzEr
[2]HArd real-time DEpendable Sytems

2.5 Bound-T

Bound-T[1] is a tool developed by Space Systems Finland mostly through ESA (European Space Agency) funded projects. Similarly to aiT, Bound-T analyzes compiled and linked executables and provides some automatic analysis of loop bounds. During the arithmetic analysis of a subprogram, Bound-T finds the potential loop counter variables for each loop and tries to bound the initial value, the step value (increment or decrement), and the limit (loop terminating) value of each potential loop counter. If it succeeds, it bounds the number of repetitions of the loop. If there are several loop counters for the same loop, Bound-T uses the one that gives the least number of repetitions. When this process does not work, the user must supply annotations.

2.6 Gromit

The Microelectronic System Design group in the Forschungszentrum Informatik an der Universitt Karlsruhe in Germany has built a WCET tool that focus on two PowerPC processors: the PPC403, the MPC750[13, 14]. This tool, called Gromit, was written in Java with some auxiliary modules in C++. It relies on bound annotations given in auxiliary files. In the HIJA project, this is being extended to use information from data flow analysis and runtime monitoring.

2.7 Modes

The WCETAn Tool from the University of York [5, 6] aims to provide portable WCET analysis for the Java programming languages by associating timing information with the Java bytecodes in the code under analysis. This tool also uses annotations to bound loops but introduces the concept of modes to give more flexibility on bounds. A given method may have a predefined number of modes, each with its own loop fixed bound. Each call point is associated with a mode. Thus the bound used is determined by the call point, though only a finite set of modes can be defined.

2.8 Analysis

Tools	annotation	automatic
CW_167	constant	no
Cinderella 3.0	constant	no
Heptane	constant	no
aiT	constant	simple loops only
Bound-T	constant	simple loops only
Gromit	constant	will use DFA results
WCETAn	modes	no

Table 1: Loop bounds handling in current Tools

As depicted in Table 1, all current tools depend on user provided loop bounds, whether flexible as with modes or constant in other tools. Even the tools which perform analysis to find constant loop bounds need annotations in most cases. Current tools pose two problems: there is no mechanism for determining the correctness of the bound given and

the bounds themselves are inflexible. Even execution modes suffer from both drawbacks, since even there only a finite set of modes can be provided.

3. METHODOLOGY

It is impossible to set constant bounds for a given method that make sense for all applications, but a realtime JAVA developer must understand the complexity of all time critical code. Part of this is to understand upon what data the number of times a loop will iterate depends. If this information is captured in a formal language, then this information can be used in two complementary ways:

- to obtain an upper bound on the iteration count of a loop in a given method, called at a given point in a program, and
- prove the correctness of the loop bound expression.

The result is that a user can provide bounds hints that are both generally applicable and are provably correct.

All methods that can be used in time critical code must terminate. There complexity must also depend on known quantities. The Java Modeling Language (JML)[21] provides a construct for providing parametrized loop bounds: `@decreasing`. Since `@decreasing` can contain an arbitrary expression instead of a simple constant to describe a loop bound, it is possible to annotate loops with bounds that are correct of every execution of the loop. Deductive formal verification can also be used to prove the correctness of the bound, so that user input can be verified. Concrete bounds for any given call point can be determined once the values of all independent variables in the `@decrease` clause are known.

Thus, determining the upper bound of a given loop, in a given method, at a given call point, can be done by determining all possible values the data that determines the concrete bound can take and take the maximum thereof. For simple bounds that depend on a static value or a statically determinable value such as the size of an array, where the array size is a constant, the set is easily determined. For more complex attributes, annotations may be needed to relate loop relevant attributes with a given object, e.g. the maximum value that `Vector.getSize()` can return with a particular instance of `Vector`.

To make this work in practice, one must limit the expression used in the `@decreases` clause. A workable approach is to limit the expression to a integer linear equation. Though this may not work for all bounds, a much larger set of useful cases can be covered than with previous methods. The steps involved are as follows:

1. write the annotations for each loop in question;
2. use deductive formal verification to prove the loop bound;
3. use data flow analysis to propagate data for loop independent variables; and
4. then combine the results with a integer linear equation solver such as lp_solve[4].

The first two steps are both hardware and call independent and need only be done once for all uses of the given method.

The tools needed for provable bound checking, KEY for deductive formal verification and aicas' data flow analysis, where designed for more complex task, but work well for provable bounds checking.

3.1 Deductive Program Verification

Deductive Program Verification is a formal method for statically proving the correctness of a program with respect to the specification of its requirements. This technique is best suited for verifying functional properties of a system, i.e., determining whether the values computed by the system agree with its functional specification or not.

In order to statically analyze a program, Deductive Program Verification symbolically executes it faithfully reflecting the semantics of the programming language. For that mathematical logic and automated theorem proving techniques are applied. The correctness of a program p is defined by means of logical formulas ϕ and ψ that express constraints on the prestate and the poststate of p. A formula $\phi \rightarrow \langle p \rangle \psi$ is valid if for every state s satisfying precondition ϕ a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds. This formula is similar to a *Hoare Triple* $\{\phi\}p\{\psi\}$ introduced by C.A.R. Hoare in 1969 [17] with the difference that in the former formula ϕ and ψ might contain programs while Hoare triples can only be expressed in terms of pure first-order logic. Thus, Hoare logic is usually not sufficient to express ϕ and ψ for real world computer programs—a special program logic is necessary, such as *Java Dynamic Logic* (JAVADL) [3] an instance of *dynamic logic* [12] used in the KEY project (`http://key-project.org/`).

The KEY tool[2], or simply KEY, is a verification tool jointly developed at Karlsruhe University (Karlsruhe, Germany), Chalmers University (Gothenburg, Sweden) and the University of Koblenz (Koblenz, Germany). It performs deductive verification of an annotated JAVA code in order to statically prove its correctness with regards to its specifications. As any analysis tool, KEY requires the properties to be analyzed to be formally specified. In KEY, functional properties specification has a form of a *contract* described in the *design by contract* (DBC) paradigm, popularized by Bertrand Meyer ([22]). DBC uses the conceptual metaphor of a legal contract. The parties in the contract are a "client" that wants to use a software component and a "supplier" that provides it. If the client satisfies its part of the contract, referred to as the *precondition*, then the supplier promises to keep his part, referred to as the *postcondition*. Furthermore there are laws governing all contracts in a given domain, referred to as *invariants*. DBC is a very general paradigm and there is no restriction on the language used to formulate invariants, preconditions and postconditions. In fact, almost anything between natural language and Lisp code has been employed for this purpose. In the HIJA project [15], the JAVA *Modeling Language* (JML) [20] is used for specifications for its closeness to the JAVA language, its syntax, and semantics. Invented by Gary Leavens, JML has now turned into a group effort supported by a growing number of researchers worldwide. JML invariants are attached to JAVA classes and JML contracts to JAVA methods. Preconditions and postconditions are expressed via *requires* and *ensures* clauses. All clauses are directly placed as specially tagged comments into the JAVA code. The syntax of JML expressions closely follows that of JAVA.

The main component of the KeY tool is the KeY prover, a semi-automated prover with a comfortable graphical user interface. Special care has been taken to make user interaction, when it is inevitable, as easy as possible, e.g. through flexible selection of proof strategies, the possibility to save and reload (partial) proofs, to rerun proofs on changed proof obligations, built-in decision procedures for integers, drag-and-drop interaction, and a clearly laid-out display of the proof. JML (or OCL[8]) specifications are automatically translated into proof obligations in JavaDL. The JavaDL calculus covers 100% of JavaCard, version 2.1, and it is continually being extended in order to support full sequential Java features (it actually supports most of them).

3.2 Data Flow Analysis

Most of the work on data flow analysis has concentrated on pointer analysis[16]. Pointer analysis typically uses static, program wide data flow analysis, which is an iterative algorithm that determines an upper bound for the set of values each reference variable in an application may hold. In addition to the set of values for each variable, the analysis determines a set of invocations, where each invocation is a method call together with context information at the call. The resulting set of invocations is an upper bound for the set of invocations that may be performed during an actual program run.

The analysis starts with an empty set of variable values and the set of invocations containing only the main routine of the analyzed application. In each iteration, the set of possible values each variable may hold is joined with the set of values that are assigned to these variables by any method that is in the invocation set. Also, any new invocation that is performed by a method that is in the set of invocations is also added to this set. The iterative analysis continues as long as these two sets grow, it stops when the smallest fix point has been reached, i.e., when the sets of values and calls remained constant during a complete iteration over all calls.

Pointer analysis may be context sensitive or context insensitive. For context sensitive analysis, the context of the caller is part of the representation of invocations and values. Context information usually is the call chain that leads to an invocation. However, context may include other information such as the thread that performs the invocation or environmental information such as the current allocation context when region based memory management is used [27, 26]. Context insensitive pointer analysis identifies invocations and values by the method that is invoked and the source code position that creates a value, respectively.

Context insensitive analysis significantly reduces the analysis complexity, but it provides results that are not accurate enough for all purposes: values created in different contexts cannot be distinguished and result in the inability to proof the absence of an error. For example, the different instances of a container structure that are used by different threads to store different objects cannot be distinguished, a context insensitive analysis cannot detect that thread local objects stored into a thread local container are not accessible by another thread that uses a different thread local container instance.

On the other hand, context sensitive analysis keeps too much information. Since the complete call chain is usually used as context in context sensitive analysis, this leads to infinite value sets for recursive routines, the call chain has to be reduced to contain no or only a limited number of cycles. However, even with this restriction, the number of possible call chains typically grows exponentially with the application size making context sensitive analysis difficult to use with real world applications.

Pointer analysis can furthermore be classified as flow sensitive and flow insensitive. A flow sensitive analysis respects the order of statements during the analysis of one routine, while a flow insensitive analysis ignores this order. A flow sensitive analysis achieves higher accuracy since information available in the control graph (such as a null pointer check) can be used to reduce the set of values of a variable. Unlike context sensitivity, the effect of flow sensitivity on the performance of the analysis is less critical. Routine wide, or global data flow analysis is a technique that is widely applied in optimizing compilers at a tolerable costs.

Object sensitive points-to analysis forJava was presented by Milanova et. al [23]. In object sensitive analysis, the context information does not consist of the call chain, but the allocation site of the current object (*this* in Java) is used as context information. This approach brings a significant improvement in accuracy compared to a context insensitive analysis while the analysis complexity grows less than using a context sensitive analysis based on the complete call chain.

In the HIJA project aicas has adapted context sensitive and flow sensitive pointer analysis to an object oriented environment. The definition of object context has been extended to include object sensitivity. Providing object contexts improves the accuracy of the analysis results in combination with a reasonable analysis space and time complexity. This analysis distinguishes object initialization and object use which significantly improves the analysis accuracy for object oriented applications. Specific properties such as singleton instances and embedded instances are also detected to further improve analysis accuracy.

The aicas' data flow analysis program can prove the absence of many error conditions. The development of the analysis algorithm was driven by the requirement to reduce the number of false positives in the set of potential errors that is reported. The error conditions include Java runtime errors such as null pointer use, illegal casts and illegal array stores, but also more complex error conditions such as absence of deadlocks and the correct use of region based memory management

using the mechanisms available in the Real-Time Specification for Java [7]. The analysis results are used for the verification of safety critical Java applications that are developed according to the profiles defined within the HIJA project. The program can also propagate data values for use with loop bound determination.

3.3 Synthesis

Once bounds are proven with KeY and information for bounds are propagates via DFA to all call sites, the results must be reduced to concrete loop bounds. For bounds that are integer linear equations this can be done with any integer linear equation solver. KeY find the initial values for loop variables. Together with the decreases clause and the data flow results, there is sufficient information for the integer linear equation solver to provide the concrete provably correct bounds for each call site.

4. EXAMPLE

An an example, one can use a method from a safety critical avionics application: the *mergeProcedures* method as depicted in Figure 1. The top of the figure shows the general view of the method: it receives two procedures and merges them producing a result procedure containing some legs from the first and some legs from the second procedure, where legs are partial paths. The way the procedures are merged is shown at the bottom of the figure. *legs1* and *legs2* represent the legs from the procedures 1 and 2 respectively (in this implementation *legs1* and *legs2* are arrays). The resultant procedure is built with legs from *legs1* (from the beginning until the point *i1*) and from *legs2* (from the point *i2* until the end). The calculation of the points *i1* and *i2* determines which legs will be part of the result procedure. These points are calculated as following.

- *i1* represents the last leg from *legs1* to be part of the result procedure. It is found by searching for an intersection point between *legs1* and *legs2*. Each leg from *legs1*, starting with the first, is tested to see whether it exists in *legs2* or not. If one exists, this legs becomes *i1*. If no intersection was found, the last leg of *legs1* becomes *i1*.
- *i2* represents the first leg from *legs2* to be part of the result procedure. Its calculation is performed within the calculation of point *i1*. When an intersection is found, the first leg of *legs2* after the intersection leg becomes *i2*. If no intersection is found, the first leg from *legs2* becomes *i2*.
- *i3* represents a limit for the search for intersection performed in *legs2*. It is the first occurrence of a leg of the type *runway*.
- The search for intersection in *legs2* is performed backwards (from the *i3* point to the beginning) in order to find the last intersection point before the runway.

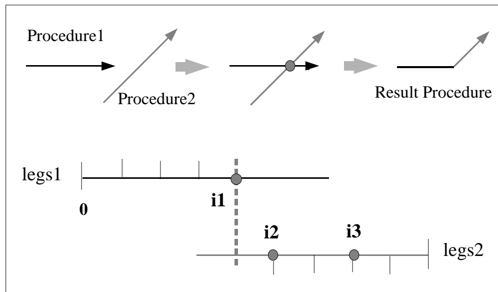

Figure 1: The mergeProcedures method analysis.

The method is a good candidate for explaining the proposed loop bound verification approach because, in spite of its small size, it contains five loops, two of them nested. It is subdivided in three submethods.

- **indexFirstRunway** calculates the point *i3*. It contains one loop over *legs2*.
- **findIntersection** calculates the points *i1* and *i2*. It contains a nesting of two loops: for each element $x1$ of *legs1* an element $x2$ is search backwards in *legs2* from the point *i3*, such that $(x1 = x2)$[3].
- **finishMerging** generates the result procedure (array). It contains two loops: (i)legs from *legs1* from the beginning until *i1* are copied to the result array, and (ii) legs from *legs1* from *i2* until the end are copied to the result array.

In order to functionally verify these methods, contracts have to be written expressing the preconditions and postconditions for each method. In addition, each loop would have also to be fully specified. In KeY, a way of specifying a loop is annotating it with the following three items.

- **Loop invariant** A loop invariant is any relation (or formula) that remains true during the whole loop execution (including immediately before starting the loop and immediately after finishing the loop). Thus many formula would be possible candidates for an loop invariant. However, in practice, one has to chose the invariant that leads to the postcondition to be proven. One should keep in mind that the only information that the KeY tool has after symbolically executing the loop is the information contained in the loop invariant. Consequently, the loop invariant has to be as strong as the method postcondition requires.
- **Assignable clause** The assignable clause is the only loop clause adopted by KeY that is not yet in JML. A similar clause is already being used by other groups, so it is just a matter of time before this clause officially becomes part of JML. It decreases the complexity of the loop proof by establishing the variables that are modified by the loop body.
- **Decreases clause** A decrease clause is a formula that must decrease by at least one in each step of the loop and remain positive during loop execution. Considering an integer expression D being the decrease clause, KeY verifies that D only takes positive values and strictly decreases by 1 until $D = 0$ when symbolically executing the loop body. The decrease clause expression implicitly defines the following invariant for the loop: $D^i \geq D \geq 0$ where $D = f(x_1, \ldots, x_n)$ and D^i is the initial value of D (immediately before the loop execution).

An example of a JML annotated Java implementation of the *findIntersection* method is shown in Fig.2. In its contracts the precondition establishes that for using the method the arrays *legs1* and *legs2* must be non null and have at least one element each and that the runway index *i3* must be within the bounds of *legs2*. In addition, the postcondition establishes the possible values and relations for the returned indexes *i1* and *i2*. Consequently, the loop invariants must be as strong as necessary for proving this postcondition.

This is a reasonable contract for proving the correctness of the method calculated values. However, when the focus goes to the verification of loop bounds instead of functional

[3]In the original application this comparison is more complex. However the difference does not have any effect in this demonstration

```
/*@ private normal_behavior
  @ requires legs1 != null && legs1.length > 0 &&
  @          legs2 != null && legs2.length > 0 &&
  @          i3 > 0 && i3 < legs2.length ;
  @ assignable i1, i2;
  @ ensures i2 == 0 && i1 == legs1.length - 1 ||
  @        (i2 >  0 && i2 <= i3 + 1 &&
  @         i1 >= 0 && i1 < legs1.length &&
  @         legs1[i1] != null &&
  @         legs1[i1] == legs2[i2 - 1]);
  @*/
 private void
   findIntersection(Leg[] legs1, Leg[] legs2, int i3)
 {
   i1 = legs1.length - 1; i2 = 0;
   int j = 0; int k = i3; Leg leg;
   /*@ loop_invariant
     @  j <= legs1.length && j >= 0 &&
     @  i2 == 0 && i1 == legs1.length - 1;
     @ assignable leg, j, k, i1, i2;
     @ decreases (legs1.length - j);
     @*/
   while (j < legs1.length)
     {
       leg = legs1[j];
       if (leg instanceof KFixLeg)
         {
           k = i3;
           /*@ loop_invariant
             @  k <= i3 &&
             @  ((k >= -1 && i2 == 0  &&
             @    i1 == legs1.length - 1) ||
             @   (j < legs1.length && j >= 0 &&
             @    k > -1 && i2 == k+1 &&
             @    i1 == j && legs1[j] !=null &&
             @    legs1[j] == legs2[k]));
             @ assignable k, i1, i2;
             @ decreases (k + 1);
             @*/
           while (k >= 0)
             {
               if (leg == legs2[k])
                 {
                   i1 = j;
                   i2 = k + 1;
                   return;
                 }
               k--;
             }
         }
       j++;
     }
   }
```

Figure 2: A findIntersection method annotated for functional verification.

behavior, the way of specifying a method can be much simpler. Since the the correctness of the calculated values is not relevant, no postcondition needs to be specified. In this case, a trivial value *true* can be used. Consequently, loop invariants can be also simplified. In fact, for the case where the decrease clause contains only one loop dependent variable (a variable declared in the assignable clause of the loop), the implicit loop invariant seems to be sufficient for proving the method termination. The implicit loop bound states that the decreases term is always positive and in every loop iteration less than the evaluation of the decreases term with the loop variable instantiated to its initial value. Since this

```
/*@ private normal_behavior
  @ requires legs1!=null && legs1.length>0 &&
  @          legs2!=null && legs2.length>0 &&
  @          i3>0 && i3<legs2.length ;
  @ assignable i1, i2;
  @ ensures true;
  @*/
 private void
   findIntersection(Leg[] legs1, Leg[] legs2, int i3)
 {
   i1 = legs1.length - 1;
   i2 = 0;
   int j = 0;
   int k = i3;
   Leg leg;
   /*@ assignable leg, j, k, i1, i2;
     @ decreases (legs1.length - j);
     @*/
     while (j < legs1.length)
       {
         leg = legs1[j];
         if (leg instanceof KFixLeg)
           {
             k = i3;
             /*@ assignable k, i1, i2;
               @ decreases (k + 1);
               @*/
             while (k >= 0)
               {
                 if (leg == legs2[k])
                   {
                     i1 = j;
                     i2 = k + 1;
                     return;
                   }
                   k--;
               }
           }
         j++;
       }
   }
```

Figure 3: A findIntersection method annotated for loop bound verification.

invariant can be generated automatically, for this case the user would not to have to write any invariant for the loop. This case covers the vast majority of cases that one would expect to find in safety critical code.

An example of the same JAVA implementation of the *findIntersection* method already presented annotated for loop bounds verification is shown in Fig.3. For the verification, the following invariants where generated automatically:

- for the outer loop, considering $D = (legs1.length - j)$, the JML loop invariant

```
(legs1.length - 0) >= (legs1.length - j) &&
(legs1.length - j) >= 0
```

- for the inner loop, considering $D = (k + 1)$, the JML loop invariant

```
(i3 + 1) >= (k + 1) && (k + 1) >=0
```

The *indexFirstRunway* method annotated for loop verification is shown in Fig.4. The loop invariant

```
(legs2.length - 0) >= (legs2.length - j) &&
(legs2.length - j) >= 0
```

```
/*@   private normal_behavior
  @   requires legs2 != null && legs2.length > 0;
  @   assignable \nothing;
  @   ensures true;
  @*/
private static int indexFirstRunwayTest(Leg[] legs2)
{
  int j = 0; Leg leg;
  /*@ assignable j, leg;
    @ decreases (legs2.length - j);
    @*/
  while (j < legs2.length)
    {
      leg = legs2[j];
      if (leg instanceof KFixLeg)
        {
          if (((FixLeg)leg).getFix() instanceof Runway);
          return j;
        }
      j++;
    }
   return legs2.length - 1;
 }
```

Figure 4: A indexFirstRunway method annotated for loop bound verification.

Method	Nodes	Branches
indexFirstRunaway	1518	24
findIntersection	2430	28
finishMerging	10599	56

Table 2: Statistics on KeY proofs.

```
/*@ private normal_behavior
  @ requires legs1 != null && legs1.length > 0 &&
  @          legs2 != null && legs2.length > 0 &&
  @          i1 >= 0 && i1 <  legs1.length     &&
  @          i2 >= 0 && i2 <= legs2.length;
  @   assignable \nothing;
  @   ensures true;
  @*/
private
  Leg[] finishMergingTest(Leg[] legs1, Leg[] legs2)
{
  int diff = i2 - i1 - 1;
  Leg[] merge =
    new Leg[(i1 + 1 + (legs2.length - i2))];
  int j = 0; int k = i1 + 1;
  /*@ assignable merge[0..i1], j;
    @ decreases (i1 - j + 1);
    @*/
  while (j <= i1)
    {
      merge[j] = legs1[j];
      j++;
    }
  /*@ assignable merge[i1 + 1..(merge.length - 1)],k;
    @ decreases (merge.length - k);
    @*/
  while (k < merge.length)
    {
      merge[k] = legs2[k + diff];
      k++;
    }
  return merge;
}
```

Figure 5: A finishMerging method annotated for loop bound verification.

was generated for this example. The *finishMerging* method annotated for loop verification is shown in Fig.5. The loop invariant

```
(i1 - 0 + 1) >= (i1 - j + 1) && (i1 - j + 1) >= 0
```

was generated for the first loop and

```
(merge.length - (end1+1)) >= (merge.length - k) &&
(merge.length - k) >= 0
```

was generated for the second loop.

The correctness of the three methods was proven using KeY and the results are shown in the Table 2. All three methods could be proved mostly automatically and where performed in last than 30 min. Just to have an idea of the improvement on time, the proof of the functional behavior of the three methods required more than 5 hours. Both times refer to the time spent using the KeY tool for proving the methods, without considering the time for writing the contracts. It is interesting to highlight that, since the contracts for functional behavior were already written and proven correct for the three methods, the time for writing contracts for loop bound verification was practically null, requiring only substituting the postcondition by true and erasing the loop invariants. In fact, even this would not be necessary, since once functional correctness is demonstrated, the loop bound proof comes for free.

The loop bound proved for each loop is the initial value of the decrease clause (previously referred as D^i). Its value is already calculated by KeY and can be provided automatically without the loop dependent variables. Once this expression is known, a DFA tool can be used to propagate the other variables in order to calculate the value of the bound to be used by other analysis, like WCETA. For instance, considering the outer loop *findIntersection* method (Fig. 3), the bound is given by the initial value of its decrease clause ($legs1.length - 0$) after the propagation from a DFA tool of the length of the array *legs1*.

5. CONCLUSION

Using deductive formal verification and data flow analysis, it is possible to obtain provably correct bounds for loops for resource analysis. This technique has two main advantages over previous techniques: bounds can be formulated to be application independent and the bounds themselves can be formally proved. The proofs themselves are much simpler than those required for full functional verifications, both in the information that needs to be specified and the difficulty of the proof. Not withstanding, the annotations are completely compatible with full functional verification, providing the user with a simple migration path.

6. FUTURE WORK

Thus far, enough work has been done to demonstrate that the technique does work. For practical application, better integration among the tools is necessary. Some of this work has begun in the HIJA project, but the synthesis of data flow analysis and deductive formal verification needs to be completed. Also, better integration with WCETA tools could significantly reduce the work required for analyzing a system by reusing the integer linear equation set between different call points of the same function. It would also be useful to conduct a large use case study to evaluate to what extend this methodology can be applied. At least some of these issues will be examined in the German SuReal project.

In the approach realized so far the term appearing in the decreases clause has to be provided by the designer. In a

next step one could imagine automated tool support for finding it. For simple loops this is already done in the aiT and the Bound-T tool. Also for this task analysing the loop body via symbolic execution could prove to be successful. Some preliminary results are contained in [11].

So far, the issue of exceptions was not addressed. To make WCET analysis useful not only verified bounds on loop iterations are required, but it should also be verified that no exceptions are throw during its execution. Deductive verification can also play an important role in doing so.

7. ACKNOWLEDGMENTS

Some of the funding for this work was provided through the HIJA project sponsored by the European Union IST Sixth Framework for which the authors are grateful.

8. REFERENCES

[1] Bound-t tool homepage. URL: `http://www.bound-t.com/`.

[2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041, pages 6–24. Springer-Verlag, 2001.

[4] M. Berkelaar. lp_solve (mixed integer linear program solver) ftp site. `ftp://ftp.es.ele.tue.nl/pub/lp_solve`. Reported to have solved models with up to 30,000 variables and 50,000 constraints. A Java version is available at `http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml`.

[5] G. Bernat. Javelin webpage. URL: `http://www-users.cs.york.ac.uk/~bernat/javelin/index.html`, Mar. 2000.

[6] G. Bernat, A. Burns, and A. Wellings. Portable worst case execution time analysis using java byte code. In *Proc. 12th EUROMICRO conference on Real-time Systems*, June 2000.

[7] G. Bollela. *Real-Time Specification for Java*. Addison-Wesley, 2001.

[8] T. Clark and J. Warmer, editors. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.

[9] A. Colin. Heptane webpage. URL: `http://www.irisa.fr/solidor/work/heptane-demo/heptane.html`, Feb. 2001.

[10] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherlands, June 2001.

[11] T. Gedell and R. Hähnle. Automating verification of loops by parallelization. *submitted*, 2006.

[12] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[13] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on modern processor architectures. In *Proceedings of the Date 2000 Conference*, pages 552–559, Paris, France, Mar. 2000.

[14] A. Hergenhan, A. Siebenborn, and W. Rosenstiel. Studies on different modeling aspects for tight calculations of worst case execution time. In *WIP-Proceedings of the 21th IEEE Real-Time Systems Symposium*, Orlando FL, USA, Nov. 2000.

[15] Hija, high-integrity java. www.hija.info, 2004-2006.

[16] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, June 2001.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[18] R. Kirner. calc_wcet_167 webpage. URL: `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`, Oct. 2001.

[19] R. Kirner. The programming language wcetc. Research Report 2/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[20] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, Nov. 2004. Draft revision 1.98.

[21] G. T. Levens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual. http://www.jmlspec.org/, 2004.

[22] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[23] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM Press.

[24] *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, Jan. 1994.

[25] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, May 1989.

[26] M. Tofte. A brief introduction to Regions. pages 186–195, 1998.

[27] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, Feb. 1997. An earlier version of this was presented at [24].