AUTOMATIC DERIVATION OF PATH AND LOOP ANNOTATIONS IN OBJECT-ORIENTED REAL-TIME PROGRAMS*

JAN GUSTAFSSON
DEPT. OF COMPUTER ENGINEERING
MÄLARDALENS UNIVERSITY, BOX 883
S-721 23 VÄSTERÅS, SWEDEN
JGN@MDH.SE
AND ANDREAS ERMEDAHL
DEPT. OF COMPUTER SYSTEMS
UPPSALA UNIVERSITY, BOX 325
S-751 05 UPPSALA, SWEDEN
EBBE@DOCS.UU.SE

Abstract. This paper presents a new method for deriving path and loop annotations automatically for object-oriented real-time programs. Such annotations are necessary when the worst case execution time (*WCET*) of programs is to be calculated. Normally these annotations must be given manually by the programmer.

By automating the the generation of annotations the user is relieved from a task that can be quite difficult and highly critical. If, for example, a programmer gives a too optimistic annotation for a loop, a tool may calculate a too short *WCET*. This may invalidate subsequent analysis, leading to that a real-time system may violate its real-time properties.

The method is based on the notion of abstract interpretation, and is illustrated by showing the analysis of an example in Smalltalk.

The paper also discusses alternatives to the semantic analysis used in the method. It is shown that syntactic analysis suffices in some cases, and that it can be used to reduce the calculation cost for loops.

Keywords: real-time software, object-oriented programming, execution time analysis, semantic analysis, syntactic analysis, abstract interpretation.

1. Introduction. To be able to guarantee the correct timely behaviour needed in hard real-time systems, the execution time of the software in the system must be known prior to execution. As the execution time of most programs varies, depending on input data and the system state, the worst case execution time, *WCET*, is used as a least upper limit on the execution time. For programs with some complexity, it can be hard to find the input data that causes the *actual* worst case execution time, $WCET_A$. Therefore, measurement is not considered a feasible method in the general case. Instead, static analysis, which from the source code derives $WCET_C$ (the *calculated* worst case execution time), has been proposed by many researchers (see for example [PK89, PS90, CBW94, LM95, PS95, Alt96]). $WCET_C$ should be as *tight* approximation of $WCET_A$ as possible to avoid waste of resources. Still, it must be *safe*, i.e., $WCET_C \geq WCET_A$.

However, a common drawback of these methods is that *manual annotations* are required. Manual annotations are extra information in the program, given to the $WCET_C$ tool. The annotations can be written as comments in a special format, or in an extension to the used language. The following example from [PK89] shows an extension of the C syntax used to give annotations for the maximum number of iterations of `for`-loops.

```
FOR (expr1; expr2; expr3) MAX_COUNT(constr_expr)
     statement1
[ON_OVERRUN(statement2)]
```

FIG. 1.1. *Example of syntax for manual loop annotations*

The expression `constr_expr` is given by the programmer and is calculated statically by the compiler to yield the maximum number of iterations of `statement1`, to be used by the $WCET_C$ tool. The number of iterations is also supervised during run-time, and if overrun occurs, `statement2` is executed. The upper-case letters mark that this is an extension to the normal C syntax.

*This work was supported by The National Board for Industrial and Technical Development (DOORS project, 93–3180) and ASTEC (Advanced Software Technology, a competence center at Uppsala University).

There are typically three types of manual annotations:

a) The *maximum number of iteration count* for loops. Calculating the iteration count is impossible in the general case, since the problem is equivalent to the well-known, undecidable, halting problem.

b) The *maximum depth of recursive function calls* for the same reason as in a). However, recursion is usually not allowed in real-time programs.

c) Information of *dependent program parts.* In this way, false paths (non-executable paths) can be identified and excluded from the analysis, thereby making tighter $WCET_C$ calculation possible. Examples of this type of annotation can be found in [Par93].

But to give these manual annotations is not always a simple task. It is easy to find small examples where, e.g., the maximum number of iterations in a loop is hard to calculate, and where the dependencies are complicated (see section 3 for an example). The main problems are that:

- To give manual annotations means *extra work.* The programmer has to figure out the worst behaviour in his loops and to find false paths.
- To give manual annotations is *error-prone.* An incorrect annotation may lead to one of two unpleasant situations:
  1. A too big (untight) $WCET_C$, causing overreservation of resources.
  2. Or worse, a too small (unsafe) $WCET_C$, causing risk for missed deadlines, with possible catastrophic consequences.

The idea of our new method is introduce an analysis that finds the annotations *automatically*, as they are inherent in the semantics of the program. Termination of the analysis is guaranteed by defining a *time budget* for the program. The analysis can be seen as a first phase of a $WCET_C$ calculation. The derived annotations are used by a following phase, an object-code analysis, which also considers modern hardware architectures with caches and pipelines.

In our analysis of a program we will use *data flow analysis* to find possible variable values at different points in the program. Using this information we can find false paths and calculate minimum and maximum iterations in loops. We will not consider recursion in this paper, but the method can be applied to recursion as well.

The method has been studied by the authors for a subset of C [EG97a]. We show by a small example in Smalltalk how the method may be extended to object-oriented languages. As shown in the example in section 3, also sizes of collections (data structures in Smalltalk) and other useful information for execution time calculation can be extracted. One of the goals of the method is to be a part of execution time calculations for RealTimeTalk (RTT), which is a Smalltalk dialect for hard real-time systems, developed at Mälardalens University [EMTP$^+$96].

The remainder of the paper is organised as follows: In section 2, the new method is introduced. In section 3 we present an illustrative example. Section 4 describes related work in the area of program analysis. Section 5 discusses ways to make loop analysis more efficient using syntactic loop analysis. Section 6 makes some conclusions, and finally, section 7 gives ideas for future work.

2. Analysis of Smalltalk programs. Smalltalk is a "pure" object-oriented programming language where everything is an object. The syntax is built on three basic mechanisms: assignments, message passing and return expressions. Smalltalk supports single inheritance and polymorphism and is dynamically typed. For a presentation of Smalltalk, we refer to [GR89].

When analyzing Smalltalk programs we are interested in the values of variables at different points in the program. We call these points *control points*, denoted by $c_0, c_1, \cdots, c_m$. With each control point c_i we associate the program code p_i, which is the rest of the code to be analyzed at that point. For the following three Smalltalk expressions we may insert control points as shown below:

$[c_0]$
```
a := 5.             [c1]
mySet := Set new.   [c2]
mySet add: a.       [c3]
```

With each control point c_i we also associate one or more *states*, σ_i^j, each of which holds the possible values of the variables at the corresponding control point at program execution. We distinguish between different passings j of a control point, since it can be passed several times in e.g., a loop.

The *concrete semantics* (meaning) of a program p_0 is defined by a transition system over sets of *configurations*, γ (we are using the terminology of [NN92]). A configuration is either *intermediate*, $\gamma = \langle p_i, \sigma_i \rangle$, or *final*, $\gamma = \sigma$. The transition relation "$\Rightarrow$" is defined using rules of structural operational semantics.

The semantics of a program is defined transitively in terms of "$\Rightarrow$" so that

$$\mathcal{P}[\![p_0]\!]\sigma_0 = \begin{cases} \sigma & \text{if } \{\langle p_0, \sigma_0 \rangle\} \stackrel{\star}{\Rightarrow} \{\sigma\} \\ \perp & \text{otherwise.} \end{cases}$$

For the initial state σ_0 a program p_0 *terminates* iff there is a finite sequence of configuration sets $\{\gamma_0\} = \{\langle p_0, \sigma_0 \rangle\} \Rightarrow \ldots \Rightarrow \{\langle \varepsilon, \gamma_k \rangle\} \Rightarrow \{\gamma_k\}$, where ε represents an "empty" program, i.e., the analysis is finished. If γ_k is a final configuration, the program has terminated *successfully.* If γ_k is an intermediate configuration, we have a *stuck* configuration, meaning there is no transition possible. This is probably due to an error in the program (like "method not found"). If the sequence is infinite, the program does not terminate (indicated with $\perp$).

We analyze sets of configurations, because some of our transitions yield more than one configuration. See figure 2.3 and figure 2.6.

If each variable in the initial state, σ_0, is assigned to one single value, like $\sigma_0 = [\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 4]$, the evaluation of the rules will correspond to a "normal" execution of the program.

For our analysis, we will define an *abstract state* where variables can be assigned to *split intervals* of integers or floats, like $\sigma_0 = [\mathtt{a} \mapsto 1..3, \mathtt{b} \mapsto 2.0..4.0 \vee 7.0]$. We can have polymorphic variables, like $\sigma_1 = [\mathtt{c} \mapsto 0..3 \vee 9.0]$, meaning that `c` is either an integer with value 1, 2 or 3, or a float with value 9.0. We can have collections (data structures) of variables, like the following collection of two elements: $\sigma_2 = [\mathtt{d} \mapsto \{3..6 \vee 9.0, 11..20\}]$. This syntax means that the first element of `d` is either an integer with the value 3, 4, 5 or 6, or a float with value 9.0, and the second is an integer with a value between 11 and 20 (inclusive). This restricted domain can in a straight-forward manner be extended to other types of data and collections.

For each concrete semantic rule in the programming language, a corresponding *abstract rule* is defined. For example, our abstract version of the '+' operator must handle split intervals of integers and reals instead of single ones.

The cost (time and memory) needed to express and calculate the exact states is often too big. The representation can then be simplified by a safe approximation. For example, $a \mapsto 1 \vee 3 \vee .. \vee 99$ can be replaced by 1..99. The approximation must be safe (possible values must not be removed), tight (as few extra values as possible), and efficient (in terms of analysis calculations). We will of course face a trade-off between cost of computation and quality of results.

Abstract interpretation techniques [CC77] can be used to define a correct relation between the abstract and the concrete domains.

2.1. Analysis of sequences. For each expression in a sequence, the state is updated with the corresponding abstract rule. If a message send is performed, the invoked method is analysed and the return values are stored in the state.

For example, consider the program in figure 2.1. Here we identify the control points c_0, c_1, c_2 and c_3 respectively, where c_0 denotes the start of the program.

```
[c0]
a := 5.          [c1]
b := b - 2.      [c2]
a := a + b.      [c3]
```

FIG. 2.1. *Smalltalk sequence*

A concrete and an abstract evaluation of the sequence can be seen in figure 2.2. We have in the concrete evaluation assumed that $a = 1$ and $b = 4$ in the initial state. In the abstract evaluation we have $a = 1..3$ and $b = 2..4 \vee 7$ as the initial state. The abstract values are either input values or values calculated during analysis of the previous code. If nothing is known about the values, they are assumed to be $\top$, e.g., for an integer any value in the interval *minint..maxint*, which is the smallest and the largest integer value possible, respectively, for the current language implementation and hardware configuration. Note that the abstract evaluation corresponds to a set of concrete evaluations and that each concrete evaluation corresponds to one execution.

$$\begin{aligned}\sigma_0 &= [a \mapsto 1, b \mapsto 4]\\ \sigma_1 &= [a \mapsto 5, b \mapsto 4]\\ \sigma_2 &= [a \mapsto 5, b \mapsto 2]\\ \sigma_3 &= [a \mapsto 7, b \mapsto 2]\end{aligned}$$

(a) Concrete

$$\begin{aligned}\sigma_0 &= [a \mapsto 1..3, b \mapsto 2..4 \vee 7]\\ \sigma_1 &= [a \mapsto 5, b \mapsto 2..4 \vee 7]\\ \sigma_2 &= [a \mapsto 5, b \mapsto 0..2 \vee 5]\\ \sigma_3 &= [a \mapsto 5..7 \vee 10, b \mapsto 0..2 \vee 5]\end{aligned}$$

(b) Abstract

FIG. 2.2. *Different evaluations of the Smalltalk sequence*

If the selector is implemented by more than one method (polymorphism), all are analysed, and the resulting states are merged (see section 2.4).

2.2. Analysis of selection constructs. Selection in Smalltalk is accomplished with `ifTrue`-constructs. The idea is to perform the analysis of the two alternate paths in an `ifTrue`-construct in *two* states, as shown in the rule in figure 2.3.

$$\{\langle \texttt{(C) ifTrue: [E1] ifFalse: [E2]. P}, \sigma\rangle\} \Rightarrow \{\langle \texttt{E1. P}, \sigma^t\rangle, \langle \texttt{E2. P}, \sigma^f\rangle\}$$
where
$$\sigma^t = \mathcal{C}[\![\texttt{C}]\!]\sigma$$
$$\sigma^f = \mathcal{C}[\![\texttt{not C}]\!]\sigma$$

FIG. 2.3. *Rule for* `ifTrue`*-constructs*

The rest of the code in the program is denoted with `P`. The expression `E1` is evaluated in the state where `C` is true, σ^t, and `E2` is evaluated where `C` is false, σ^f. A condition can be seen as a constraint to be applied on the variables in a given state. The construct $\mathcal{C}[\![\texttt{C}]\!]\sigma$ constrains a state σ with the condition `C` to yield the constrained state.

2.2.1. False paths . If the abstract value of a variable in a state is empty (i.e., $\perp$), the corresponding path is considered as false. This is the way our method finds false paths.

Figure 2.4 contains an example of an false path. The program has a path that is false for *all* input data (assuming that `S1` and `S2` does not change the value of `a`).

```
(a > 10) ifTrue:  [S1] ifFalse: [S2].
(a > 5)  ifTrue:  [S3] ifFalse: [S4]
```

FIG. 2.4. *Example 1 of false paths*

We see that the path passing the expressions `S1` and `S4` is a false path, since $a > 10$ implies $a > 5$. Please note that the program does *not* contain dead code, since the statements `S1` and `S4` are executed in other paths.

We can also have false paths which depend on input data. For example, in the program in figure 2.5, the path passing the expression `a := a + 1` is false, since the state where this statement should be evaluated is empty. In this case, we must assume that a previous analysis has resulted in a possible value range where $a \geq 2$. In this example, the false paths analysis also identifies dead code, since the expression `a := a + 1` never will be executed.

```
"a := 5..10."
(a < 2) ifTrue: [a := a + 1]
```

FIG. 2.5. *Example 2 of false paths*

2.3. Analysis of loop constructs. The core idea is to transform loops into `ifTrue`-constructs. For example, consider the `whileTrue`-expression. We will analyse it according to the rule in figure 2.6.

$$
\begin{array}{l}
\{\langle \texttt{[C] whileTrue: [S]. P}, \sigma\rangle\} \Rightarrow \\
\quad \{\langle \texttt{S. [C] whileTrue: [S]. P}, \sigma^t\rangle, \{\langle \texttt{P}, \sigma^f\rangle\} \\
\qquad \text{where} \\
\qquad\quad \sigma^t = \mathcal{C}[\![\texttt{C}]\!]\sigma \\
\qquad\quad \sigma^f = \mathcal{C}[\![\texttt{not C}]\!]\sigma
\end{array}
$$

FIG. 2.6. *Rule for* `whileTrue`*-constructs*

The `ifTrue`-construct generates two states each time it is analysed, as defined in the previous section. After each iteration we will have:

1. The state in which the loop will continue, σ^t.
2. The state in which the loop will terminate, σ^f.

We can find minimum and maximum number of iterations by keeping track of the iteration count. This can be made by introducing a loop counter for each loop during analysis, which is set to 0 before the loop and which is incremented each iteration. We can record the following two events:

- Minimum occurs where $\sigma^f \neq \bot$ for the first time in the next iteration.
- Maximum occurs where $\sigma^t = \bot$ for the first time in the next iteration.

When $\sigma^f = \bot$ the loop *must* continue. So, when $\sigma^f \neq \bot$ for the first time, it *may* terminate, and the minimum number of iterations has been identified.

Similarly, when $\sigma^t \neq \bot$, the loop *may* continue. So, when $\sigma^t = \bot$ for the first time, it *must* terminate, and the maximum number of iterations has been identified.

Since we are using approximations in our analysis, as mentioned earlier, the states used during analysis may contain values that are not possible in a real execution. This may lead to a loop state σ^f becoming $\neq \bot$ too early in the analysis, leading to a too low minimum number of iterations. Similarly, the loop state σ^t may become $= \bot$ too late, leading to a too high maximum number of iterations. Still, the computed values are safe.

A loop is "rolled out" until it terminates, or until the time budget is exceeded (see section 2.5). An example is provided in section 3.

2.4. Merging of states. Each `ifTrue`-construct will double the number of states when it is analysed, and every `whileTrue`-expression will double the number of states each iteration, as explained above. To avoid exponential complexity explosion, states can be *merged* at certain points, so the analysis can continue from *one* state instead of several. Merging is allowed when all configurations are at the same control point.

$$
\begin{array}{l}
\{\langle p, \sigma^1\rangle, \langle p, \sigma^2\rangle, \cdots, \langle p, \sigma^n\rangle\} \Rightarrow \{\langle p, \sigma\rangle\} \\
\quad \text{where} \\
\qquad \sigma = \sigma^1 \oplus \sigma^2 \oplus \cdots \oplus \sigma^n
\end{array}
$$

FIG. 2.7. *The effect of merging*

The value of a variable in the merged state $\sigma = \sigma^1 \oplus \sigma^2$ is the union of its values in σ^1 and σ^2. Collections are merged elementwise. The merging may lead to overestimation of possible values, depending on how states are represented and where and how merging is performed. As a first heuristics, we have decided to merge configurations at four points in the analysis:

1. After polymorphic message sending.
2. After each loop iteration.
3. When a loop has terminated.
4. When a method has terminated.

In this way we will avoid a state explosion, to the price of introducing some pessimism, both in terns of overestimation and that false paths are only discovered on one level of code within methods or loop bodies. As always, we have a trade-off between quality of results and calculation costs.

In the example in section 3, merging is done after each loop iteration, and when the loop has terminated. Merging points are marked with *oplus* in the figures 3.2 and 3.4.

2.5. Termination of the analysis. To guarantee termination of the analysis we will use the fact that a real-time program must terminate within a given time [LL73]. This *time budget*, T_{budget}, should be a realistic upper time limit for the program on a given hardware. The time budget may be calculated during the design phase and can be seen as part of the specification of the program. It is the only "annotation" needed by our the method. Note that our time annotation is different from the annotations of other methods. Erroneous path or loop annotations may lead to a wrong $WCET_C$, but an erroneous T_{budget} may, in the worst case, only lead to a failure of the analysis, not to an incorrect result.

Each construct or program block has an associated time interval $t_{minc}..t_{maxc}$, its minimum and maximum execution time. This time is calculated from the corresponding assembler code. With our own compiler for RTT, we have full knowledge of the generated code. If the hardware uses caches, pipelines etc., the time intervals will increase. In this case the time budget may have to be increased.

For each analysed construct, the corresponding time interval will be added to an accumulated time for the analyzed path. The longest path, with its interval $T_{minc}..T_{maxc}$, is compared to the time budget during the analysis. If $T_{maxc} \leq T_{budget}$ holds when the analysis is finished, we can guarantee that the program will not exceed its time budget. If $T_{maxc} > T_{budget}$, the program may not terminate within the time budget, and the analysis should stop.

3. Example. The Smalltalk program in fig. 3.1 will be analysed by our the method. The program uses the `SmallInteger` class with the methods `<=`, `<`, `+` and `*`, the `Float` class with the methods `+` and `*`, the `OrderedCollection` class with the method `add`, assignments, the `True` and `False`-classes for selection constructs (`ifTrue:ifFalse:`) and a loop construct (`whileTrue:`). The example contains a selection within a loop, where a collection is filled with variables of different types. The example does not illustrate inheritance or recursion.

We will compare the analysis results to the real results, in terms of overestimations of numbers, sizes, and values. We assume that the initial values are $n = 1..10$ and $x = 2.0..7.0$.

```
"n := 1..10. x := 2.0..7.0. Initially"              [c0]
myOC := OrderedCollection new.                      [c1]
[n <= 5] whileTrue: "Start of loop" [               [c2]
   (n < 3)  ifTrue: [                               [c3]
       myOC add: (n * (n + 1))] "Integer"           [c4]
             ifFalse: [                             [c5]
       myOC add: (x * (x + 2))]. "Float"            [c6]
   n := n + 2.                                      [c7]
   x := x + 1.                                      [c8]
   ]. "End of loop"                                 [c9] [c10] [c11]
```

FIG. 3.1. *The example program*

In the program we have inserted control points after each expression plus two extra for merging:

- c_9 for merging of the loop body ($\sigma_9^j = \sigma_8^{jt} \oplus \sigma_8^{jf}$). In the example, σ_i^{jf} denotes the environment at control point c_i at loop iteration j, with a false selection condition. Similarly, σ_i^{jt} denotes the same control point and loop iteration with a true selection condition.

Iter.	Loop cond.	If cond.	Control point	σ	`n`	`x`	`myOC`	Note
0			c_0	σ_0	1..10	2.0..7.0		
0			c_1	σ_1	1..10	2.0..7.0	{}	
1	f		c_{10}	σ_{10}^1	6..10	2.0..7.0	{}	1. min # iter = 0
1	t		c_2	σ_2^1	1..5	2.0..7.0	{}	
1	t	t	c_3	σ_3^{1t}	1..2	2.0..7.0	{}	
1	t	t	c_4	σ_4^{1t}	1..2	2.0..7.0	{2..6}	
1	t	t	c_7	σ_7^{1t}	3..4	2.0..7.0	{2..6}	
1	t	t	c_8	σ_8^{1t}	3..4	3.0..8.0	{2..6}	
1	t	f	c_5	σ_5^{1f}	3..5	2.0..7.0	{}	
1	t	f	c_6	σ_6^{1f}	3..5	2.0..7.0	{8.0..63.0}	
1	t	f	c_7	σ_7^{1f}	5..7	2.0..7.0	{8.0..63.0}	
1	t	f	c_8	σ_8^{1f}	5..7	3.0..8.0	{8.0..63.0}	
1	t		c_9	σ_9^1	3..7	3.0..8.0	{2..6 $\vee$ 8.0..63.0}	2. $\sigma_9^1 = \sigma_8^{1f} \oplus \sigma_8^{1t}$
2	f		c_{10}	σ_{10}^2	6..7	3.0..8.0	{2..6 $\vee$ 8.0..63.0}	
2	t		c_2	σ_2^2	3..5	3.0..8.0	{2..6 $\vee$ 8.0..63.0}	
2	t		c_3	σ_3^{2t}	$\perp$			3. $\sigma_3^{2t} = \perp$
2	t	f	c_5	σ_5^{2f}	3..5	3.0..8.0	{2..6 $\vee$ 8.0..63.0}	
2	t	f	c_6	σ_6^{2f}	3..5	3.0..8.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0}	
2	t	f	c_7	σ_7^{2f}	5..7	3.0..8.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0}	
2	t	f	c_8	σ_8^{2f}	5..7	4.0..9.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0}	
2	t		c_9	σ_9^2	5..7	4.0..9.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0}	4. $\sigma_9^2 = \sigma_8^{2f} \oplus \perp$
3	f		c_{10}	σ_{10}^3	6..7	4.0..9.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0}	
3	t		c_2	σ_2^3	5	4.0..9.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0}	
3	t		c_3	σ_3^{3t}	$\perp$			5. $\sigma_3^{3t} = \perp$
3	f	f	c_5	σ_5^{3f}	5	4.0..9.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0}	
3	f	f	c_6	σ_6^{3f}	5	4.0..9.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0, 24.0..99.0}	
3	f	f	c_7	σ_7^{3f}	7	5.0..10.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0, 24.0..99.0}	
3	f	f	c_8	σ_8^{3f}	7	5.0..10.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0, 24.0..99.0}	
3	f		c_9	σ_9^3	7	5.0..10.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0, 24.0..99.0}	6. $\sigma_9^3 = \sigma_8^{3f} \oplus \perp$
4	t		c_2	σ_2^4	$\perp$	5.0..10.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0, 24.0..99.0}	7. max # iter = 3
4	f		c_{10}	σ_{10}^4	7	5.0..10.0	{2..6 $\vee$ 8.0..63.0, 15.0..80.0, 24.0..99.0}	
4			c_{11}	σ_{11}^4	6..10	2.0..10.0	{} $\vee$ {2..6 $\vee$ 8.0..63.0} $\vee${2..6$\vee$ 8.0..63.0, 15.0..80.0} $\vee${2..6 $\vee$ 8.0..63.0, 15.0..80.0, 24.0..99.0}	8. result after analysis: $\sigma_{11}^4 = \sigma_{10}^1 \oplus \sigma_{10}^2 \oplus \sigma_{10}^3 \oplus \sigma_{10}^4$

FIG. 3.2. *Abstract evaluation of the example program*

- c_{11} for merging of all environments (at c_{10}) where the loop has terminated ($\sigma_{11}^4 = \sigma_{10}^1 \oplus \sigma_{10}^2 \oplus \sigma_{10}^3 \oplus \sigma_{10}^4$).

Figure 3.2 shows the analysis of the example program. From left to right, the columns show the iteration count, the values of the loop and selection conditions, the control point, the environment and the abstract values of `n`, `x` and `myOC`.

- Note 1 marks the minimum number of iterations (0) in the loop. It is found where $\sigma^f \neq \perp$ the first time in the next iteration.

`n` (start)	`n` (end)	`x` (start)	`x` (end)	Iter.	`myOC` (end)
1	7	2.0..7.0	5.0..10.0	3	{2, 15.0..80.0, 24.0..99.0}
2	6	2.0..7.0	4.0..9.0	2	{6, 15.0..80.0}
3	7	2.0..7.0	4.0..9.0	2	{8.0..63.0, 15.0..80.0}
4	6	2.0..7.0	3.0..8.0	1	{8.0..63.0}
5	7	2.0..7.0	3.0..8.0	1	{8.0..63.0}
6	6	2.0..7.0	2.0..7.0	0	{}
7	7	2.0..7.0	2.0..7.0	0	{}
8	8	2.0..7.0	2.0..7.0	0	{}
9	9	2.0..7.0	2.0..7.0	0	{}
10	10	2.0..7.0	2.0..7.0	0	{}
Total:	6..10		5.0..10.0	0..3	{} $\vee$ {8.0..63.0}$\vee$ {6, 15.0..80.0}$\vee$ {8.0..63.0, 15.0..80.0}$\vee$ {2, 15.0..80.0, 24.0..99.0}

FIG. 3.3. *Actual possible environments after execution*

- Note 2, 4 and 6 marks merging of the loop body.
- Note 3 and 5 marks false paths (when the selection condition true yields $\sigma^t = \perp$ in the loop body at iteration 2 and 3).
- Note 7 marks the termination of the loop. It is found when the true loop condition yields an empty environment, i.e., $\sigma_2^4 = \perp$, implying that the maximum number of iterations in the loop is 3.

Figure 3.4 shows the evaluation graphically, from the start state σ_0 to the resulting state σ_{11}^4. The index of the merging function $\oplus$ refers to one of the four types of merging described in section 2.4.

The purpose of the example is to show that the analysis gives results which can be used, e.g., to annotate the program graph. When we compare the result of the analysis (note 8 in figure 3.2) to the real values (in figure 3.3), we can conclude the following:

1. *Number of iterations in loops.* Minimum number of iterations is 0 and maximum is 3. No overestimation is made in the analysis.
2. *False paths and dead code.* We have found false paths (in iterations 2 and 3) but no dead code (since the code is executed in iteration 1).
3. *Size of collections.* Minimum size of `myOC` is 0 and maximum is 3 elements. No overestimation is made in the analysis.
4. *Number of different types in polymorphic collections.* We have two (safe) overestimations. First, only one integer does not occur. In the maximum case, the analysis gives one integer and two floats, or three floats. This is the second overestimation, since the latter case does not occur.
5. *Possible values for all variables in each control point in the program.* The variables `n` and `x` are estimated correctly. However, the collection `myOC` is (safely) overestimated, since none of the collections {2..5, 15.0..80.0} and {3..6, 15.0..80.0, 24.0..99.0} occur after execution.

The results in point 1 and 2 are useful in $WCET_C$ calculation, as mentioned in section 1. The results in point 3 and 4 can be useful if the code in figure 3.5 follows the example program.

In the $WCET_C$ calculation of the code, the size of `myOC` is useful (for number of iterations) and the type of the elements (since floats take more time to calculate than integers). Finally, the result in point 5 gives values useful in subsequent analysis.

4. Related Work. The purpose of the method described in this paper is to calculate annotations to be used in $WCET_C$ calculation. This is accomplished by static program analysis based on abstract interpretation. This seems to be a novel idea, even though there is an extensive body

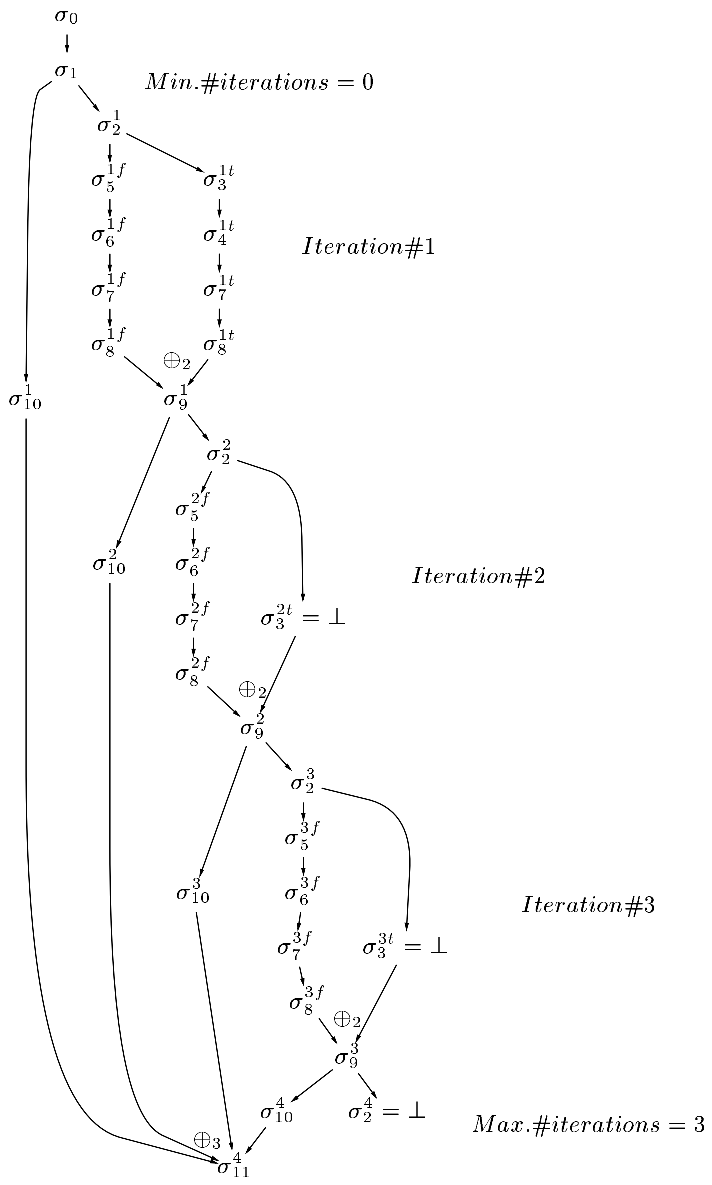

FIG. 3.4. *Graph showing the abstract evaluation*

of research in the program analysis area, but with other aims, like compiler optimizations, program proofs, and language construction.

The method in this paper derives an approximation of variable values in the control points of Smalltalk/RTT programs. A number of researchers have done similar analyses for other languages, like C and Pascal (see below). No one has analysed Smalltalk in this way, however.

4.1. Abstract Interpretation. Cousot and Cousot laid the theoretical framework for abstract interpretation in [CC77]. Abstract interpretation can be seen as a non-standard semantics, in which the domain of values is replaced by a domain of descriptions of values, and in which the operators are given a corresponding non-standard interpretation. A number of researchers have used methods based on abstract interpretation in program analysis, primarily in the context of compiler optimizations. Their efforts have been to find value ranges by static analysis to yield more effective and smaller programs.

```
myOC do: [:el | "For each element do"
 (el < 100) ifTrue: [
   el := el^2] " take the square"
 ifFalse: [
   el := el^3]. "or take the cube"
].
```

FIG. 3.5. *Program fragment*

Harrison showed as early as 1977 [Har77], albeit not using the abstract interpretation framework, that interval range analysis could be used to statically calculate possible values for variables. The results could be used to eliminate redundant tests, choose data representation, and more.

In a more recent paper, [VCH96], Verbrugge et. al. uses abstract interpretation to statically analyze the full C language (including pointers, functions and recursion) to yield a range for each variable at each point.

Bourdoncle uses abstract interpretation and interval range analysis for "abstract debugging" [Bou93]. He tries to find potential errors in programs by calculating values of variables. In this way, he can identify possible division by 0, indeces out of bounds, and more. He has developed a prototype tool, Syntox, which analyses Pascal programs.

Patterson describes an analysis similar to the above used for static branch prediction in [Pat95]. The results from his analysis are useful in compiler optimizations, especially for instruction scheduling, cache optimizations and register allocation. Although he doesn't refer to Cousot and Cousot, the method is very similar to abstract interpretation with interval range analysis. The value range representation is more detailed than the common interval $l..u$, however. Patterson uses split intervals, where each part interval, beside a lower bound l and an upper bound u, also has a probability and a stride (arithmetic step size).

His method can find some kinds of false paths, namely those where dead code is found (like the one in figure 2.5). It cannot find false paths like the one in figure 2.4, because his method merges values after each if-statement.

His method tries to optimize loop analysis by comparing the operations made on variables in loops to different templates. If simple increments are found, for example, the template can be used to derive the final value range from the initial value and the increments.

It should be clear from the paper that we use a different method than the above-mentioned methods based on abstract interpretation. In those methods, a loop is seen as a system of equations, where the union of the input and output values of the loop are used as input to the loop. The loop is then "iterated" until a fixed-point is found. This means that in each control point in the loop, we have the *union* of the possible values of all iterations of the loop. (The technique is described in more detail in, e.g., [Bou93]). In our method, the loop is "rolled out" (see for example the rule in figure 5.1), and each loop iteration is analysed *separately.* We will calculate the possible values of the values in the control points of each iteration of the loop. In this way, we will have a much more exact approximation, albeit to a higher calculation cost.

4.2. Symbolic Execution. There are also a few methods based on *symbolic execution.* For example, Altenbernd can find false paths automatically for some simple cases (see [Alt96]). Forsyth uses symbolic execution, program annotations and a notion of execution modes in [For92]. With his method, false paths can be excluded and the $WCET_C$ values that are calculated are sensitive to both input data and program modes. However, both Altenbernds and Forsyths methods rely on manual annotations.

5. Complexity of Loop Analysis. One problem that program analysis has to face is the potential risk of "getting stuck" in loops. The approach used in the method presented in this paper will analyse each loop iteration that is possible during run-time, and for each iteration create two states. It is obvious that there is a risk that a tool could be too slow and memory gobbling to be used for programs containing complex, nested loops with many iterations.

5.1. Widening/Narrowing Techniques. A common way to solve this in similar methods, is to use *widening/narrowing* techniques, as devised by Cousot and Cousot, and used in e.g., [Bou93]. The idea is first to use a widening function that gives a safe (upper) approximation of the fixed points of the loop values. Next, a narrowing function is used to find a safe approximation of the least fixed point. The right widening and narrowing functions can give a significant speed-up of the convergence of the loop analysis.

Now, there are some problems with the widening/narrowing approach, which makes it unsuitable for our method:

- As we don't use fixpoint computations for loops, the widening/narrowing method doesn't fit into our current approach.
- It is not easy to find the right widening and narrowing functions. They must be carefully selected "by hand" and may be hard to include in a general tool.
- The narrowing step normally only narrows down the loop index. Other variables calculated in the loop are often pushed up to infinity by the widening function, and are never narrowed down by the narrowing function. This is far too crude for our analysis, if we need those values in the subsequent analysis.

5.2. Syntactic Loop Analysis. Another approach would be to try to to find the number of iterations in loops by *syntactic analysis* (to differ from *semantic loop analysis*, the main topic of this paper). There are actually some simple cases where it is possible to statically decide the minimum and maximum number of iterations from the syntax of the loop construct. There are even unterminating loop constructs which we can identify. This is similar to what Patterson is doing in [Pat95]. Blieberger is studying this problem in [Bli94], where he defines a new kind of loop, *discrete loops*, which are more general than for-loops, but still analyzable.

Of course, syntactic analysis cannot handle all cases, so we have to rely on semantic analysis in the general and more complex cases. Syntactic analysis should be regarded as an optimization of the analysis.

General *while-loops* are complex to analyze, but there is a number of simple cases analyzable by syntactic analysis. Let us first look at a general while-loop in Smalltalk (figure 5.1).

```
[condition] whileTrue:
      [loop body]
```

FIG. 5.1. *Smalltalk while-loop*

The block containing the loop body will be invoked as long as the condition block yields the value `true`.

There are of course an abundance of cases that a tool has to recognize and deal with. We will just look at a few, to get an idea of what is possible. Assume k and c are integer constants, $c > 0$ and that the integer loop variable i has the initial value i_0 before entering the following loop:

```
(i < k) whileTrue:  [i := i+c]
```

A mathematical analysis based on the semantics of the `whileTrue`-expression gives the following *recurrence equation* when $i_0 < k$:

$$i = i_0;$$
$$i_{n+1} = i_n + c;$$

Note that n denotes the number of iterations in the loop, because a new value for i is calculated once for each loop iteration.

A recurrence equation describes a value in terms of earlier ones, plus a boundary value. Recurrences are thorougly described in [GKP89]. The recurrence equation above has an **open form**, i.e., one has to compute all previous valus to yield the current value. A solution to a recurrence has a **closed form** which gives the value by a single computation. The solution to the recurrence above is simply

$$i_n = i_0 + nc$$

giving the following result for the two possible cases, after solving for n in the second case:

$$n = 0 \text{ and } i = i_0 \text{ if } i_0 \geq k$$
$$n = \lceil \frac{k-i_0}{c} \rceil \text{ and } i = i_0 + nc \text{ if } i_0 < k$$

So we can calculate the number of iterations exactly and quickly in this special case, assuming we can recognize it in a tool. If the loop body contains other code than just manipulating the loop variable (as it should do in a real program), the following must not occur within the loop body for the result to hold:

1. The execution encounters a return expression.
2. The index is manipulated.
3. There are program errors (like "method not found") that causes the program to fail.
4. The execution loops forever in an non-terminating loop.

The first and second requirements are relatively easy to check by a tool. The third requirement can be fulfilled by a type inference/type checker for Smalltalk (see for example [GPMTB95]). The fourth requirement can be handled if the inner loop is analyzable by syntactic analysis. If it has to be analysed by semantic loop analysis (according to section 2.3), it must be analyzed in each iteration of the outer loop, and little speed-up is gained.

As we don't execute the body in a syntactic loop analysis, we will not get the possible values of the variables updated in the body (except for the value of the condition expression or loop index). This is of course a serious drawback of the analysis, if we need those values for the analysis after the loop or for inner loops.

However, we can calculate the values for those variables by syntactic loop analysis, if we can find a formula for their values as we leave the loop. This is possible if the variables are updated in the same way that we can handle for the loop variable. (This is similar to the notion of *induction variables* in compiler theory: a variable is called an induction variable of a loop if it is incremented or decremented by some constant every time it changes value [ASU86]).

We can also find non-terminating loops by syntactic analysis. A simple example of this is the following loop:

```
(i > k) whileTrue:  [i := i+c]
```

giving the following result, after a similar analysis as above:

$$n = 0 \text{ and } i = i_0 \text{ if } i_0 \leq k$$
$$n = \perp \text{ and } i = \perp \text{ if } i_0 > k$$

As above, the assumptions 1 – 4 above has to be checked in a non-trivial loop body for the result to hold.

6. Conclusions. We have presented a new method which can be used to derive annotations for paths and loops automatically. The purpose of the method is to relieve the real-time programmer from the diffucult and error-prone task of determining annotations manually. The method also gives a number of spin-off results, like safe estimations of possible values of variables, sizes and contents of collections, and discovery of dead code.

The method is demonstrated for a subset of Smalltalk. We have omitted recursion and most classes and primitives that are included in a Smalltalk implementation. We are convinced that our method can be extended to cover also these parts of Smalltalk. We can not allow the class hierarchy to be changed during runtime, however, since this could change the semantics of the programs.

Because the method is based on the general notion of abstract interpretation, it can be adapted to other object-oriented languages, like C++, Ada95 and Java.

A potential problem is state growth during analysis of complex loops. In order to limit the calculation cost, we have described some ways to make approximations. We have also pointed out a method, syntactic loop analysis, that can be used to make analysis of simple loops more efficient.

7. Future work. Future work includes defining a detailed abstract semantics for the analysis, based on a concrete semantics of Smalltalk [MTPG97], and to develop a prototype tool. Special consideration will be taken to complexity and how the method scales.

An interesting question is: how are hard real-time programs written "out there" in the real world? How complicated constructs are the programmers really using? Do they avoid certain features that make programs harder to analyze? What aspects of $WCET_C$ calculation should we give priority to?

The authors are conducting an investigation in Swedish industry to find out what the reality looks like. An inquiry has been answered by 50+ companies working with software for embedded systems (see [EG97b]). The main goal of the investigation was to find out how the companies calculate $WCET_C$ today, and their attitude towards a $WCET_C$ tool (what functions should be included etc.). A part of the questionnaire included questions about the C language; could certain language constructs be removed to gain more analyzable programs? The inquiry is followed up by a number of interviews with software designers and programmers.

We think that we can discern two different streams in the real-time programming area.

- The first is a trend towards simple and highly modulized software, especially for embedded systems and smaller control systems. The real-time operating system allows the software to be broken down in many small, simple and relatively independent modules. Potentially dangerous features like recursion, function pointers and dynamic allocation of memory (in C) are avoided or even forbidded. Complex while-loops are avoided and replaced by simpler for-loops with a fixed number of iterations.
- The second is a trend towards more and more complex real-time systems and applications, still with hard real-time demands. We can see this trend in larger control systems, multimedia and games.

To start with, we will concentrate on software of the first type. In such systems, complexity is a smaller problem than in the second type. Maybe, syntactic analysis may even be sufficient for parts of the loop analysis.

Further investigation will show how these trends develop and inspire new ideas for the program analysis and our $WCET_C$ calculation research.

REFERENCES

[Alt96] P. Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the Eight Euromicro Workshop on Real-Time Systems*, pages 102–107, June 1996.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*, chapter 10, Code optimizations. Addison - Wesley, 1986.

[Bli94] J. Blieberger. Discrete loops and worst case performance. *Computing Languages*, 20(3):193–212, 1994.

[Bou93] F. Bourdoncle. Abstract debugging of high-order imperative languages. In *Proceedings of SIGPLAN'93 Conference on Programming Language design and Implementation*, pages 46–55, 1993.

[CBW94] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *ACM Sigplan Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[EG97a] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *EuroPar'97, workshop 20, Real-Time Systems and Constraints, Passau, Germany*, Aug 1997.

[EG97b] A. Ermedahl and J. Gustavsson. Enkät om realtids industrins syn på verktyg för exekveringstidsanalys, (inquiry towards the real-time industry of its view of tools for execution time analysis). Technical report, Department of Computer Systems, Uppsala University, Sweden, 1997. In swedish.

[EMTP+96] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, J. Gustafsson, K. Sandström, and E. Brorson. An overview of RTT - a design framework for real-time systems. *The Journal of Parallel and Distributed Computing*, 36, 1996.

[For92] C. Forsyth. Implementation of the worst-case execution analyser. *York Software Engineering Ltd*, 1992.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics – A foundation for Computer Science.* Addison - Wesley, 1989.

[GPMTB95] Jan Gustafsson, Kjell Post, Jukka Mäki-Turja, and Ellus Brorsson. Benefits of type inference for an object-oriented real-time language. In *WOORTS'95, the Workshop on Object-Oriented Real-Time Systems at the Seventh IEEE Symposium on Parallel and Distributed Processing, San Antonio, Texas, USA*, Oct 1995.

[GR89] A. Goldberg and D. Robson. *Smalltalk 80, The Language.* Addison - Wesley, 1989.

[Har77] W. Harrison. Compiler analysis of the value range of variables. *IEEE Transactions on Software Engineering*, SE-3, 1977.

[LL73] C. Liu and J. Leyland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, pages 46 – 61, Jan 1973.

[LM95] Y-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM Workshop on Lang., Comp. and Tools for RTS*, May 1995.

[MTPG97] J. Mäki-Turja, K. Post, and J. Gustafsson. An operational semantics for Smalltalk. Department of Computer Engineering, Mälardalen University, Sweden, Feb 1997.

[NN92] H. R. Nielson and F. Nielson. *Semantics with Applications.* John Wiley & Sons, 1992.

[Par93] C.Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time System*, 5:31–62, 1993.

[Pat95] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN'95 Conference on of Programming Language Design and Implementation (PLDI'95)*, pages 67–78, 1995.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):159 –176, Sep 1989.

[PS90] C.Y. Park and A.C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Proceeding of 11th IEEE Real-Time Systems Symposium*, pages 72–81, Dec 1990.

[PS95] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Report, Techn. Univ., Inst. für Technische Informatik, Vienna, April 1995.

[VCH96] C. Verbrugge, P. Co, and L. Hendren. Generalized constant propagation i C. In *Proceedings of the 6th International Conference on Compiler Construction, CC'96*, April 1996.