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

Andreas Ermedahl
Dept. of Computer Systems
Uppsala University, Box 325
S-751 05 Uppsala, Sweden
ebbe@docs.uu.se

Abstract

This paper will present a new method, based on the notion of abstract interpretation, that can be used to derive path and loop annotations automatically for object-oriented real-time programs. Normally, these annotations, necessary to get a tight calculation of the worst case execution time ($WCET$), must be given manually by the programmer.

The method is illustrated by showing the analysis of an example in Smalltalk.

Keywords: *real-time software, object-oriented programming, execution time analysis.*

1 Introduction

The execution time of most programs varies, depending on the input data and the system state. 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 [PK89, PS90, CBW94, LM95, PS95, Alt96]. $WCET_C$ should be a close approximation of $WCET_A$ to avoid waste of resources. However, a common drawback of these methods is that *manual annotations* are required. There are typically three types of manual annotations:

a) The *maximum number of iteration count* for loops. This is because loops are possibly non-terminating. To calculate the iteration count is impossible in the general case, since the problem is equivalent to the well-known 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 *dependant program parts*. In this way, false paths (non-executable paths) can be identified and excluded from the calculation, thus making the $WCET_C$ as tight as possible. (However, methods based on symbolic execution can find some dependencies automatically [Alt96]).

But to give these manual annotations is an error-prone burden. It is easy to find simple 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). An incorrect manual annotation may lead to an erroneous $WCET_C$, ending up in a missed deadline, with possible catastrophic consequences.

The idea of this new method is to let the analysis try to find the annotations *automatically*, as they are inherent in the semantics of the program. Termination of the analysis of non-terminating programs is guaranteed by defining a *time budget* for the program.

In our analysis of a program we will use *data flow analysis* [Mor86] to find out 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. As shown in the example, also sizes of collections and other useful information for execution time calculation can be extracted.

The method is earlier studied by the author for a subset of C [EG97]. We show by a small example in Smalltalk how the method may be extended to object-oriented languages. One of the goals of the method is to be a part of execution time calculation for RealTimeTalk (RTT), which is a Smalltalk dialect for hard real-time systems [EMTP+96].

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

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. Finally, section 4 gives some conclusions and ideas for future work.

2. Data flow analysis and program semantics

When we are 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 $c_0, c_1, \cdots, c_m$. For the following three Smalltalk expressions we may insert control points as shown:

```
[c0]
 a := 5.  [c1]
 mySet := Set new.  [c2]
 mySet add: a.  [c3]
```

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

The *concrete semantics* (meaning) of a program is defined in terms of the environments that can be generated by the program [NN92]. We will use a semantic function, $[\![\cdot]\!]$, that takes an environment, σ, and a $rule$ on how to modify (or constrain) the environment and returns a modified environment, σ', i.e. $\sigma' = [\![rule]\!]\sigma$. Depending on the nature of the rule, i.e., if it is a statement or a condition, we will further subdivide $[\![\cdot]\!]$ into $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{C}[\![\cdot]\!]$, where $\mathcal{S}$ stands for *statement* and $\mathcal{C}$ for *condition* rule.

If each variable in the initial environment, 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 environment* 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 $\sigma_1 = [\mathtt{c} \mapsto 0..3 \vee 9.0]$ and collections of variables, like the following collection of two elements: $\sigma_2 = [\mathtt{d} \mapsto \{3..6 \vee 9.0, 11..20\}]$. 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 exact environments 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

Sequences of expressions are handled quite straightforward. For each expression, the environment is updated with the corresponding abstract rule. If a message send is performed, the invoked method is analysed and the return values are stored. If the selector is implemented by more than one method (polymorphism), all are analysed, and the resulting environments 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* environments, as shown in the following rule:

$\mathcal{S}[\![\texttt{(C) ifTrue: [E1] ifFalse: [E2]}]\!]\sigma = \{\sigma_2, \sigma_4\}$
where

$$\begin{aligned}\sigma_1 &= \mathcal{C}[\![\texttt{C}]\!]\sigma\\ \sigma_2 &= \mathcal{S}[\![\texttt{E1}]\!]\sigma_1\\ \sigma_3 &= \mathcal{C}[\![\texttt{not C}]\!]\sigma\\ \sigma_4 &= \mathcal{S}[\![\texttt{E2}]\!]\sigma_3\end{aligned}$$

The expression `E1` is evaluated in the environment where `C` is true, and `E2` is evaluated where `C` is false. A condition can be seen as a constraint to be applied on the variables in a given environment. If the abstract value of a variable in an environment is empty (i.e., $\perp$), the corresponding path is considered as false. For example, in the program below, the path c_2 - c_3 is false, since the environment at c_2 is empty.

```
[c0]
 "a := 5..10."  [c1]
 (a < 2) ifTrue: [  [c2]
   a := a + 1.  [c3]
   ]  [c4]
```

In this way, false paths and even dead code (non-executable code) can be found.

2.3 Analysis of loop constructs

The core idea is to transform loops into `if`-constructs. For example, consider the `whileTrue`-rule:

$\mathcal{S}[\![$`[C] whileTrue: S`$]\!]\sigma =$
  $\mathcal{S}[\![$`(C) ifTrue: [S. [C] whileTrue: S]`$]\!]\sigma$

The `if`-construct generates two environments each time it is analysed (as before):

1. The environment where the loop continues, σ_{true}.
2. The environment where the loop terminates, σ_{false}.

We can find minimum and maximum number of iterations:

1. Minimum occurs where $\sigma_{false} \neq \perp$ the first time.
2. Maximum occurs where $\sigma_{true} = \perp$ next iteration.

A loop is "rolled out" until it terminates, or until the time budget is exceeded (see section 2.5). For an example, please see section 3.

2.4 Merging of environments

Each `ifTrue`-construct will double the number of environments when it is analysed, and every `whileTrue`-expression will double the number each iteration, as explained above. To avoid exponential complexity explosion, environments can be *merged* at certain points, so the analysis can continue from *one* environment instead of several. In our split interval domain, the value of a variable in the merged environment $\sigma = \sigma_1 \bigcup \sigma_2$ 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 environments are represented and how merging is performed. As always, quality of results and calculation cost are related. In the example in section 3, merging is done after each loop iteration, and when the loop has terminated.

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 in the method. Note that our time annotation is different from the annotations of other methods. Erroneous path or loop annotations may lead to wrong $WCET_C$, but an erroneous T_{budget} may, in the worst case, only lead to too early ending of the analysis.

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 cache, 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

A subset of Smalltalk is used in the example program in fig. 1 to illustrate the method. It 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.

```
"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]
```

Figure 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 ($\sigma_9^i = \sigma_8^{it} \bigcup \sigma_8^{if}$). In the example, σ_i^{jf} denotes the environment at control point c_i at loop iteration j, with a false selection condition.
- c_{11} for merging of all environments (at c_{10}) where the loop has terminated ($\sigma_{11} = \bigcup \sigma_{10}^i$).

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. $\sigma_9^1 = \sigma_8^{1f} \bigcup \sigma_8^{1t}$
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. $\sigma_9^2 = \sigma_8^{2f} \bigcup \sigma_8^{2t}$
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. $\sigma_9^3 = \sigma_8^{3f} \bigcup \sigma_8^{3t}$
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

Figure 2. Abstract evaluation of the example program

Figure 2 shows the analysis of the example program, using the new method. 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 in the loop. It is found where the loop condition is false the first time.
- Note 2, 4 and 6 marks merging of the loop body.
- Note 3 and 5 marks false paths (when the selection condition is true in the loop body at iteration 2 and 3).
- Note 7 marks the termination of the loop, found when the loop condition yields an empty environment for the true-path ($\sigma_2^4 = \perp$).

The purpose of the example was 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 2) to the real values (in figure 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 4 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 is always useful in subsequent analysis (as mentioned in section 1).

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

Figure 4. Program fragment

4 Conclusions and future work

We have presented a new method which can be used to derive annotations for paths and loops automatically. 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. In order to limit the calculation cost, we have described some ways to make approximations.

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.

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	$\{\} \vee \{8.0..63.0\} \vee \{6, 15.0..80.0\} \vee \{8.0..63.0, 15.0..80.0\} \vee \{2, 15.0..80.0, 24.0..99.0\}$

Figure 3. Actual possible environments after execution

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.

[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.

[EG97] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par'97, workshop 20, Real-Time Systems and Constraints, Passau, Germany*, Aug 1997.

[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.

[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.

[Mor86] E. Morel. Data flow analysis and global optimization. In B. Lohro, editor, *Methods and tools for compiler construction, an advanced course*, pages 289 – 315. Cambridge university press, Cambridge, 1986.

[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.

[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.