Integrating Symbolic Worst-Case Execution Time Analysis with Aspect-Oriented System Development *

Aleksandra Tešanović
Jörgen Hansson
Linköping University
Department of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

Dag Nyström
Christer Norström
Mälardalen University
Department of Computer Engineering
Västerås, Sweden
{dag.nystrom,christer.norstrom}@mdh.se

Abstract

Increasing complexity in development of real-time systems accompanied by the demand for enabling their configurability requires the integration of aspect-oriented software development with real-time system development. Since software technology for building real-time systems has to support predictability in the time domain, methods and tools for analyzing temporal behavior of the aspect-oriented software are needed. Knowing worst-case execution time is of primary importance for timing analysis of real-time systems. We contribute by providing support for predictable aspect-oriented software development, by enabling symbolic worst-case execution time analysis of the aspect-oriented software systems.

1 Introduction

Aspect-Oriented Software Development (AOSD) has emerged as a new principle for software development, and is based on the notion of separation of concerns [2]. The use of aspects in system development allows high reusability, and configurability of software systems. Increasing complexity in development of real-time systems, and the demand for enabling their configurability has opened the door for the application of new software technologies, such as AOSD. However, due to specific demands of real-time systems, applying AOSD is not straightforward, since the correctness of real-time systems depends both on the logical result of the computation, and the time when the results are produced, expressed explicitly as temporal constraints [5]. Thus, one of the most important elements when determining temporal behavior of real-time systems is the worst-case execution time (WCET) analysis, computing bounds of the execution times of the tasks in the system [4]. Hence, to be able to apply AOSD in real-time system development, we need to provide ways of analyzing temporal behavior of aspects as the development process of real-time systems has to be based on software technology that supports predictability in the time domain.

*This work is supported by ARTES (A network for Real-time and graduate education in Sweden).

We contribute by providing support for predictable AOSD, by enabling WCET analysis of aspects, components, and the resulting aspect-oriented software (when aspects are weaved into components). The WCET analysis we introduce into AOSD is based on the symbolic WCET analysis [1]. We present a new algorithm for calculating high-level WCETs of aspect-oriented software. We use this algorithm as a basis for development of a new tool, called aspect analyzer, which provides automated support for the WCET analysis of the aspect-oriented software.

The paper is organized as follows. In section 2 we present preliminaries and identify some of the problems when applying aspect-orientation in real-time system development. In section 3 we outline our approach for WCET analysis of the aspect-oriented real-time software. Paper finishes with the main conclusions in section 4.

2 Background

AOSD considers components and aspects as two different entities where aspects are automatically weaved into functional behavior of components in order to produce the overall system [3]. We have encountered two main problems in applying AOSD in real-time system development. First, compatibility of components is an important issue as every aspect cannot be weaved into every component, and all aspects cannot be compiled together. Hence, compatibility of components and aspects is a concern that needs to be accounted in a component model. Second, existing aspect languages and tools to implement aspects in real-time environment, and weave them in the component code, lack support for the aspect and the component temporal information. For these reasons we have developed a real-time component model (RTCOM), presented in figure 1. Following is a brief description of RTCOM (a detailed description can be found in [6]).

The component can be viewed as consisting of mechanisms and a policy framework. The policy framework is a set of component methods, and represents component behavior implemented by the methods, i.e., it represents the basic component functionality. Methods utilize underlying component mechanisms. The policy framework of the

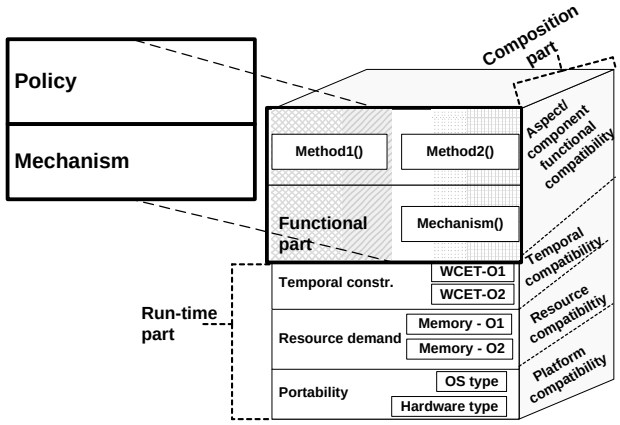

Figure 1. The component model that supports predictable aspect weaving

component can be changed or modified by aspect weaving. In this component model, aspects are policies, and, once weaved into the functional part of the component, can change component policy. The model also reflects the nature of the development process: first, mechanisms of components are implemented together with the policy framework, and then policy aspects are weaved into the components. Thus, the process results in components colored with aspects.

The component model also reflects different types of aspects we have identified in real-time systems. Beside the type of the aspects that crosscut the code of the component, we have identified run-time aspects, reflecting temporal attributes of the component, and composition aspects reflecting the compatibility between different components and aspects. With this component model we can model the behavior of aspects and components, record their compatibility and run-time properties, which are necessary for combining different aspect and components into a resulting real-time system.

3 Aspect Analyzer

As mentioned, the importance of time is emphasized in real-time system development, and one of the most important elements in real-time system development is timing (WCET) analysis of the real-time software. Determining the WCET of the code provides guarantees that the execution time will not exceed the WCET bound. We are implementing a tool, called aspect analyzer, designed to provide support for predictable AOSD. The aspect analyzer takes WCETs of the functional components and aspects, and weaves them to obtain the WCET of the resulting component or the resulting system (once aspects have been weaved in). We have developed a new algorithm, based on the notion of symbolic WCET [1], for aspect-level WCET analysis used by the aspect analyzer to weave the WCETs of aspects and components into the WCET of the system. The symbolic WCET technique describes WCET as a symbolic expression, rather than a fixed constant, in order to provide tighter bounds on the execution time. Hence, the symbolic expression is a function of a variable. A simple example: for different values of `n` in the loop (`for=1 to n`) program would have different execution times, so the WCET is a function of `n`, e.g., `WCET(n)=2ms+n*0.1ms`.

3.1 Guidelines and Assumptions

The problem we are solving is to determine the WCET of the method of a component, or a system, when aspects are weaved into the component(s) or the system. To enable efficient WCET analysis of the aspect-oriented real-time system, we need to restrict our component and system model, and weaving assumptions. Thus, components and aspects need to conform to the following:

- Each component provides mechanisms used by the methods in the component to implement certain behavior of the component. This initial behavior of the component (before any aspects are weaved) we call policy framework.
- Mechanisms are basic and fixed part of the component infrastructure and cannot be changed by aspects. Mechanisms represent fine granule method calls or function calls in the component. The internal WCET of the mechanisms is obtained by low-level code analysis, and is typically a constant number.
- Methods in the component represent coarse granule method calls. Methods are flexible parts of the component, since their implementation can be modified by weaving different aspects[1]. The internal WCET of the method (is also assumed to be known, and) can be described as a symbolic expression. The internal WCET of the method is the WCET of the code in the method excluding the WCETs of the mechanism calls.
- The *around advice* in an aspect can change the behavior of the policy framework by changing the method implementation (as it is executed instead of the code in the method). *Before advices* and *after advices* change the behavior of the policy framework without changing the original behavior of the method (as they are executed before or after the code in the method). The advices use underlying components' mechanisms to change the behavior of the policy framework. Nested advices are not allowed. We assume that the internal WCET of the code in the advices is known, and that it can be described as a symbolic expression.
- Each method and each advice declares its usage of the mechanisms, i.e., how many times it calls (uses) each mechanism.

3.2. Outline of the Algorithm

The algorithm used by the aspect analyzer for determining the WCET of the method weaved with aspects is shown

[1]Note, methods are also implemented using mechanisms, which are fixed parts of the component.

```
methodWcet( method )
1:    methodWcet = 0
2:
3:    for every advice_i in the aspect_k modifying the method
4:       do
5:       if around advice
6:          then
7:          methodWcet = methodWcet+ codeBlockWcet( advice_i )
8:       if before or after advice
9:          then
10:         methodWcet = methodWcet + codeBlockWcet( advice_i ) +
11:                      codeBlockWcet( method )
12:
13:   if method calls other methods
14:      then
15:      for every method_k called from method
16:         do
17:         methodWcet = methodWcet + methodWcet( method_k )
18:
19:   return methodWcet


codeBlockWcet( codeBlock )
1:    codeBlockWcet = intCodeBlockWcet
2:
3:    for every mechanism_i used by the codeBlock
4:       do
5:       codeBlockWcet = codeBlockWcet + wcet_i*N_i
6:
7: return codeBlockWcet
```

Figure 2. The algorithm for aspect-level WCET analysis

in the figure 2. The input to the algorithm are WCET specifications of mechanisms, policy framework, and aspects (see examples on figures 3, 4, 5, respectively). As values of internal WCETs for methods in the policy framework, and aspects can be symbolic expressions, the aspect analyzer should, in the initial step of the analysis (before it applies the algorithm), detect all the variables used in symbolic expressions, and prompt users for their values. The algorithm presented in pseudocode consists of two functions: (i) `methodWcet()`, which is a main part of the algorithm, and (ii) `codeBlockWcet()`, which is called from the `methodWcet()`.

`codeBlockWcet()` is used for calculating the WCET of a code block (`codeBlock`), which can be either an advice or a method (note that advice and methods use mechanisms as basic blocks). `codeBlockWcet()` does so by first calculating the internal WCET of a given code block in line 1. If the internal WCET (`intCodeBlockWcet`) is a symbolic expression, its value is calculated based on the symbolic expression and the value of a variable provided by the user in the initial step of the analysis. To obtain a correct WCET of a `codeBlock`, the internal value of the WCET is augmented with the values of WCET for each mechanism used by the `codeBlock` (lines 3-5), such that the value of WCET of a mechanism (which is a fixed number) is multiplied with the number of times the `codeBlock` uses the mechanism (line 5).

`methodWcet()` computes the WCET of a method that has been modified by weaving of aspects. For every advice within the aspect that modifies a certain method we need to recalculate the WCET, depending if the advice modifying the method is around, before or after. The WCET of an around advice is calculated directly by a `codeBlockWcet()`, where around advice now is a code

```
mechanisms(){
        mechanism{
                name createNode;
                wcet 5ms;
        }
        .....
        mechanism{
                name getNextNode;
                wcet 2ms;
        }

        mechanism{
                name linkNode;
                wcet 3ms;
        }
        .....
}
```

Figure 3. WCET specifications of the mechanisms in the component

block which has the internal WCET and utilizes a number of mechanisms (lines 5-7). The WCETs of before and after advices are calculated by taking into account not only the WCET of an advice as a code block, but also the WCET of the method, since the advice runs before or after the method (lines 8-11). If the method for which we are recalculating the WCET calls other methods, then in the WCET of the method we need to include all the WCETs of every other method called (which are calculated by the same principle). Thus, we need to have a recursive call to the `methodWcet()` itself (lines 14-17).

In the next section we illustrate our approach on a concrete example. We selected a simple example to be able to clearly explain the way algorithm works.

3.3 An Example

We consider a component that implements an ordinary linked list. The mechanisms implemented in the component are the ones needed for the manipulation of nodes in the list, i.e., `createNode, deleteNode, getNextNode, linkNode, unlinkNode`. We assume that the WCETs of the mechanisms are known and can be specified as showed in the figure 3. Figure 4 shows the policy framework specification of the linked list component. Each method in the framework is named and its internal WCET (`intMethodWcet`), and the number of times it uses a particular mechanism are declared. `intMethodWcet` might be represented as a symbolic expression. In this example, as the maximal number of nodes can vary influencing the execution time of the code, the internal WCETs of methods may be represented as a function of the maximal number of nodes in the list, i.e., `noOfElements` (see figure 4). Note, the value of the variable `noOfElements` is obtained from the user in the initial step of the analysis, and different values of `noOfElements` will result in different values of the WCET for the method.

Since we know the WCET of every mechanism, the WCETs of methods can be determined by simply adding up the `intMethodWcet` of the method and the WCET of every mechanism the method calls, times the number of calls to the mechanism. For example, `listInsert` method calls `createNode` and `linkNode` mechanisms

```
methods(noOfElements){
        method{
                name listInsert;
                uses{
                        createNode 1;
                        linkNode    1;
                }
                intMethodWcet 5ms;
        }
        method{
                name listRemove;
                uses{
                        getNextNode noOfElements;
                        unlinkNode 1;
                        deleteNode 1;
                }
                intMethodWcet 4ms + noOfElements*0.5ms;
        ......
}
```

Figure 4. WCET specification of the policy framework of the component

```
aspect  priorityList{

    before(noOfElements){
        method{
                name listInsert;
                        uses{
                                linkNode 3;
                                getNextNode noOfElements;
                        }
        }
        intAdviceWcet  1ms + noOfElements*0.4ms
        }
}
```

Figure 5. Specification of the aspect temporal behavior

only once, to create and link a new node in the list in the first-in-first-out (FIFO) order. The resulting WCET of the method `listInsert` would then be: `listInsertWcet=5+1*5+1*3=13`ms. Weaving an aspect into a component results in a change of the component policy, thus changing the WCET of the component. We use the aspect analyzer and the developed algorithm to determine the new WCET of the resulting code. For example, if we weave an aspect that changes FIFO policy of the component into priority-based policy (e.g., an aspect named `priorityList`). The aspect `priorityList` intercepts the method `listInsert`, and using the component mechanisms puts the node into the list based on its priority, thus changing the WCET of the component. Hence, every aspect should specify what part of the policy framework it changes. Figure 5 presents the specification of the aspect WCET. Aspect `priorityList` changes the FIFO policy into priority-based, by utilizing several component mechanisms, e.g., `getNextNode` and `linkNode`. For each advice type the changes that the advice makes to the policy framework should be specified (see figure 5). In this example, the before advice changes the WCET of the overall component. The code in the advice is executed before the code in the `listInsert` method, to ensure that the position of the node would be based on its priority. The temporal information of the aspect code now includes the internal WCET of an advice that modified the method (`intAdviceWcet`), and the information of the mechanisms used in the advice, as well as the number of times the advice has used a particular mechanism. The aspect analyzer parses WCET information, based on the algorithm presented, and computes the new WCET of the component, or the individual method call modified by weaving of the policy aspect. In this example, the resulting WCET of the method `listInsert` with the aspect `priorityList` weaved into it would be:
`priorityListInsert` = `priorityListWcet+listInsertWcet` = `intAdviceWcet+3*linkNodeWcet` + `noOfElements*getNextNodeWcet+ listInsertWcet=1+noOfElements*0.4+ 3*3+noOfElements*2+13`.
As can be seen, for different number of elements, this would result in different values of WCET.

4 Summary

In this paper we have presented a new algorithm for analyzing the worst-case execution time (WCET) of aspect-oriented software systems. The methodology we used for developing the algorithm is based on the symbolic WCET analysis of real-time software systems. To be able to efficiently analyze aspects when weaved into components, we have restricted the weaving model, and the join point model. Although there is a significant body of research done in the area of WCET for real-time software, to the best of our knowledge, this is the first work that focuses on integrating WCET analysis with AOSD, thus gearing towards predictable aspect-oriented software. As these are initial stages of the work on predictable AOSD, there are several open research issues, and we are focusing on the following: relaxing some of the assumptions for aspect weaving, and integrating the aspect analyzer with one of the existing aspect weavers, so that the analysis of the system can be performed at aspect weaving time.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th IFAC Workshop on Real-Time Programming*, Palma, Spain, May 2000.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[3] C. Lopes, E. Hilsdale, J. Hugunin, M. Kersten, and G. Kiczales. Illustrations of crosscutting. In *Proceedings of the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*, June 2000.

[4] P. Puschner and A. Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, May 2000.

[5] J. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.

[6] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. COMET: a COMponent-based Embedded real-Time database. Technical report, Dept. of Computer Science, Linköping University, and Dept. of Computer Engineering, Mälardalen University, 2002.