Aspect-Level WCET Analyzer: a Tool for Automated WCET Analysis of a Real-Time Software Composed Using Aspect and Components*

A. Tešanović*, J. Hansson*, D. Nyström†, C. Norström†, and P. Uhlin*

*Linköping University
Dept. of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

†Mälardalen University
Dept. of Computer Engineering
Västerås, Sweden
{dag.nystrom,christer.norstrom}@mdh.se

Abstract

Increasing complexity in development of real-time systems requires the integration of new software engineering techniques, such as aspect-oriented and component-based software development, with real-time system development. Since software technology for building real-time systems has to support timeliness, methods and tools for analyzing temporal behavior of the software composed out of components and aspects are needed. We contribute by providing a tool that enables automated worst-case execution time analysis of different configurations of aspects and components.

1 Introduction

Increasing complexity in development of real-time systems accompanied by the demand for enabling their configurability requires the integration of aspect-oriented and component-based software development with real-time system development. We have developed an approach to aspectual component-based real-time system development (ACCORD) [8] that integrates the two software engineering techniques, aspect-oriented and component-based software development, into real-time system development. ACCORD introduces a real-time component model (RTCOM) that provides explicit support for aspect weaving, while enforcing information hiding, i.e., it preserves basic ideas from component-based and aspect-oriented software development.

Since software technology for building real-time systems has to support timeliness [5], methods and tools for analyzing temporal behavior of the software composed out of components and aspects are needed. It is well-known that the worst-case execution time (WCET) is of primary importance for timing analysis of real-time systems.

We contribute by providing a tool that enables automated WCET analysis of different configurations of aspects and components. The tool is based on aspect-level WCET analysis [7]. The main goal of a tool for aspect-level WCET analysis is determining the WCETs of different real-time system configurations consisting of aspects and components before any actual aspect weaving (system configuration) is performed, and, hence, help the designer of a configurable real-time system to choose the system configuration fitting the WCET needs of the underlying real-time environment without paying the price of aspect weaving for each individual candidate configuration. If necessary, i.e., if very precise WCET estimates are needed, the tool for aspect-level WCET analysis can be followed by further analysis of the resulting weaved code using a more specialized WCET tool (e.g., that performs both low level and high level WCET analysis).

The paper is organized as follows. Section 2 gives the background information about RTCOM. The main constituents of automated aspect-level WCET analysis, including aspect-level WCET specifications and the aspect-level WCET analyzer, are described in section 3. Finally, in section 4 we discuss limitations and benefits of the current implementation of the tool.

2 Background

RTCOM consists of three parts: (i) functional part, which can be modified by aspect weaving, (ii) run-time part describing the run-time behavior, e.g., WCETs, of components and aspects, and (iii) composition part describing the composition rules of components and aspects. Detailed description of RTCOM can be found in [8], and here we present a brief overview of its functional part which represents the actual code of the component.

To enable efficient temporal analysis of components weaved with aspects and facilitate structured aspect weaving, while preserving information hiding, RTCOM assumes the following for the functional part (i.e., the actual code) of the component.

- Each component provides a set of mechanisms, which are basic and fixed parts of the component infrastructure. Mechanisms can be viewed as fine-granule methods or functions.
- Each component provides a set of operations to other components and/or to the system. The implementation

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

of the operations determines the initial behavior of the component, i.e., the policy framework. Operations can be viewed as coarse-granule methods or functions. Operations are implemented using the underlying mechanisms, which are fixed parts of the component.

Existing aspect languages can be used for implementing aspects and integrating (weaving) them into the functional part of RTCOM. Aspect weaving is done by the aspect weaver corresponding to the aspect language [3]. In an aspect language each aspect declaration consists of advices and pointcuts. A *pointcut* consists of one or more join points, and is described by a pointcut expression. A *join point* refers to a point in the component code where aspects should be weaved, e.g., a method, or a type (struct or union). An *advice* is a declaration used to specify the code that should run when the join points, specified by a pointcut expression, are reached. Different kinds of advices can be declared, such as: (i) *before advice*, which is executed before the join point, (ii) *after advice*, which is executed immediately after the join point, and (iii) *around advice*, which is executed in place of the join point.

Each aspect, as prescribed by RTCOM, is implemented using a number of mechanisms of a component and represent a (new) component policy. Implementation of an aspect is not limited only to mechanisms of one component as the same aspect can influence several components, and, thus, can be implemented using the mechanisms from a number of components. Weaving of aspects into the code of a component does not change the implementation of mechanisms, only the implementation of operations within the component. Hence, aspect weaving changes the policy of the component by changing one or more operations, and changing the number of mechanisms used by a particular operation.

Consider a simple example of an ordinary linked list implemented based on RTCOM. The mechanisms needed are the ones for the manipulation of nodes in the list, i.e., `createNode`, `deleteNode`, `getNextNode`, `linkNode`, and `unlinkNode`. Operations implementing the policy framework, e.g., `listInsert`, `listRemove`, `listFindFirst`, define the behavior of the component, and are implemented using the underlying mechanisms. In this example, `listInsert` uses the mechanisms `createNode` and `linkNode` to create and link a new node into the list in first-in-first-out (FIFO) order. Hence, the policy framework is FIFO.

Assume that we want to change the policy of the component from FIFO to priority-based ordering of the nodes. Then, this can be achieved by weaving an appropriate aspect. Figure 1 shows the `listPriority` aspect, which consists of one pointcut `listInsertCall`, identifying `listInsert` as a join point in the component code (lines 2-3). When this join point is reached, the code in the `before` advice `listInsertCall` is executed. Hence, the aspect `listPriority` intercepts the operation (a method or a function call to) `listInsert`, and before the code in `listInsert` is executed, the advice, using the component mechanisms (`getNextNode`), determines the position of the node based on its priority (lines 5-14).

```
aspect listPriority{
1:
2: pointcut listInsertCall(List_Operands * op)=
3:   call("void listInsert(List_Operands*)")&&args(op);
4:
5: advice listInsertCall(op):
6:   void before(List_Operands * op){
7:     while
8:        the node position is not determined
9:      do
10:       node = getNextNode(node);
11:       /* determine position of op->node based
12:       on its priority and the priority of the
13:       node in the list*/
14:    }
15: }
```

Figure 1. The `listPriority` aspect

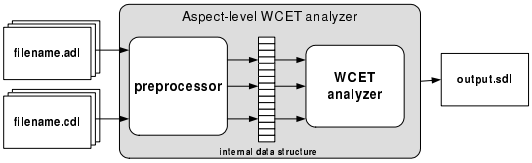

Figure 2. The overview of the automated aspect-level WCET analysis

3 Aspect-Level WCET Analysis

Figure 2 presents an overview of the automated aspect-level WCET analysis and its main constituents, including: (i) the input files that are aspect-level WCET specifications of aspects and components, (ii) the aspect-level WCET analyzer that consists of the preprocessor and the WCET analyzer, and (iii) the output files that represent the result of aspect-level WCET analysis.

The following sections give the description of aspect-level WCET specifications and the internals of the aspect-level WCET analyzer.

3.1 Aspect-Level WCET Specifications

Aspect-level WCET specifications are inputs to the aspect-level WCET analyzer, and they are defined by the run-time part of RTCOM [8].

Based on the description of the functional part of RTCOM (see section 2), the following can be observed: (i) aspect weaving does not change temporal behavior, i.e., WCETs, of mechanisms, and (ii) aspect weaving changes temporal behavior, i.e., WCETs, of operations, by changing the number of mechanisms that an operation uses. Therefore, if the WCETs of mechanisms are known and fixed, and the WCETs of the policy framework and aspects are given as a function of the mechanisms used, then the WCET of a component weaved with aspect(s) can be computed by calculating the impact of aspect weaving to WCETs of operations within the component (in terms of mechanism usage).

Thus, aspect-level WCET specifications that are fed into the tool consist of WCET specifications of the components, i.e., policy framework, and the aspects as a function of mechanism used. In the representation of WCET specifications we utilize the notion of symbolic WCET analy-

```
policy(noOfElements){
  operation{
    name listInsert;
    uses{
      createNode 1;
      linkNode   1;
    }
    intWcet 1ms;
  }
  operation{
    name listRemove;
    uses{
      getNextNode noOfElements;
      unlinkNode  1;
      deleteNode   1;
    }
    intWcet 4ms;
  }
....
}
```

```
mechanisms{
  mechanism{
    name createNode;
    wcet  5ms;
  }
  mechanism{
    name linkNode;
    wcet  4ms;
  }
  mechanism{
    name getNextNode;
    wcet  2ms;
  }

....
}
```

Figure 3. The WCET specification of the policy framework

```
aspect listPriority(noOfElements){
  advice{
    name listInsertCall;
    type before;
    changes{
      name listInsert;
        uses{
          getNextNode noOfElements;
        }
    }
    intWcet 4ms+0.4*noOfElements;
  }
....
}
```

Figure 4. The WCET specification of the `listPriority` aspect

sis [1]. Hence, we assume that the WCETs of the mechanisms are known and can be specified by symbolic expressions. Furthermore, the policy framework WCET specification consists of the mechanism usage of each operation in the framework, and the internal WCET specification. Similarly, the WCET specification of an aspect describes: (i) the type of each advice within the aspect, (ii) operations an advice modifies, (iii) the usage of mechanism by the advice while modifying the operations, and (iv) the internal WCET of the advice. The internal WCET of the operation/advice is the WCET of the code in the operation/advice excluding the WCETs of the mechanism calls. We assume that the internal WCETs are known and can be expressed in a form of a symbolic expression.

Figure 3 presents an instance of a WCET specification for the policy framework of the linked list component. Each operation in the framework is named and its internal WCET (`intWcet`) with the number of times it uses a particular mechanism are declared (see figure 3). The WCET specification for the aspect `listPriority` that changes the policy framework is shown in figure 4. Since the maximum number of elements in the linked list can vary, the WCET specifications are parameterized with the `noOfElements` parameter.

Aspect-level WCET specifications are currently implemented such that an aspect-level WCET specification of a component is contained in a file that has the extension *cdl* (component description language), while aspect-level WCET specifications of aspects have the extension *adl* (aspect description language). Our tool for aspect-level WCET analysis outputs a file with an extension *sdl* (system description language) that contains all the operations of the components in the configuration of the real-time system under analysis, and their respective resulting WCETs.

3.2 The Aspect WCET Analyzer

The aspect-level WCET analyzer consists of two parts: the preprocessor and the WCET analyzer. The preprocessing step is needed to extract the WCET information contained in the input files in a form usable by the WCET analyzer. Hence, the preprocessor takes aspect-level WCET specifications of aspects and components of a real-time system configuration as an input. It analyzes the WCET specifications given and produces data structures that store: (i) values of internal WCETs for operations and advices, (ii) values of WCETs of mechanisms, (iii) parameters existing in the symbolic expressions of operations, mechanisms, and advices, and (iv) dependency information, e.g., the mechanisms used by an operation, and advices modifying an operation. These data structures are internal to the aspect-level WCET analyzer and they are coupling the preprocessing and the analyzing part of the tool (see figure 2). The preprocessor is implemented using Bison [2] and Flex [4].

Since internal WCETs in the aspect-level WCET specifications are symbolic expressions, the values of these need to be determined, and the fist step is to obtain the values of parameters in the expressions. This is done by the aspect-level WCET analyzer in the step before invoking the WCET analyzer. The global function `checkParameters()` of the aspect-level WCET analyzer checks the data structures created in the preprocessing step detecting the parameters of operations, mechanisms, and advices (used in symbolic expressions), and prompts the human user for their values. The resulting parameterized data structures are then used by the WCET analyzer as an input to calculate the WCETs of all operations within the real-time system configuration under development.

The WCET analyzer performs the actual aspect-level WCET analysis. It does so based on the resulting parameterized data structures obtained in the preprocessing step of the analysis, and the algorithm for aspect-level WCET we developed previously [6, 7]. The algorithm provides a set of rules that define how to compute a new WCET of an operation weaved with aspects, depending on the type of an advice in the aspect. For example, for the advice of the type before modifying an operation, the new WCET of the operation would be computed using the value of an old WCET (i.e., WCET of an operation without aspects), and augmenting that value with the WCET of the before advice. This rule reflects the fact that the code of the before advice would, after aspect weaving, be inserted before the code of the operation. Similar rules exist for the advices of types after and around. Following the example of the linked list component, we can compute the WCET of the operation `listInsert` modified with an advice `listInsertCall` of the type before as follows.

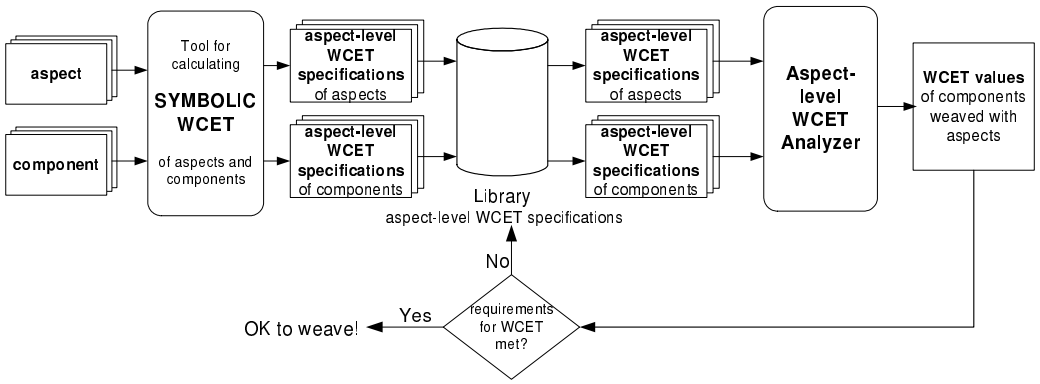

Figure 5. An overview of the aspect-level WCET analysis lifecycle

(aspectualized)listInsertWcet
= listInsertWcet(without aspects) + (before)listInsertCallWcet
= 14 + 2.4*noOfElements

where

listInsert(without aspects)
= intWcet + Σ mechanism*usage
= 1 + createNodeWcet*1 + linkNodeWcet*1
= 1 + 5*1 + 4*1 = 10

(before)listInsertWcet
= intWcet+ Σ mechanism*usage
= 4 + 0.4*noOfElements+ getNextNodeWcet*noOfElements
= 4+2.4*noOfElements

4 Limitations and Benefits

Ideally, the complete process of the aspect-level WCET analysis should have a lifecycle as presented in figure 5. The process starts with the implementation files of components and aspects, which are fed into a tool that performs the symbolic WCET analysis on the code, i.e., computes symbolic expressions for WCETs, and extracts these into aspect-level WCET specifications. These specifications are stored in a library and are used by the aspect-level WCET analyzer. Based on the output of the aspect-level WCET analyzer, i.e., computed values of the WCETs of a real-time system configuration consisting of components and aspects, we can determine the configuration eligibility for use in the underlying real-time environment with respect to WCET constraints of the environment. If a given configuration does not fulfill the requirements with respect to the WCET, the designer can choose another configuration, i.e., another set of aspect-level WCET specifications, until the WCET requirements are met, and the actual weaving can be performed.

Figure 5 also illustrates limitations of current automated aspect-level WCET analysis. The tool that computes WCETs in the form of symbolic expressions and extracts these to aspect-level WCET specifications should be an adaptation of the tool for symbolic WCET analysis to the aspect-level WCET analysis. The current implementation of the aspect-level WCET analyzer works only with aspect-level WCET specifications.

The current implementation, although given the limitations, provides benefits over traditional WCET analysis performed on weaved code since it enables calculations on WCET specifications, not on actual components and aspects. This way we reduce the overhead of performing the weaving and then WCET analysis for each potential configuration of aspects and components. Additionally, aspect-level WCET analysis can be generalized beyond symbolic WCET analysis if another approach (or a tool) for WCET analysis is used for determining the internal WCETs of operations, mechanisms, and advices.

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] C. Donnely and R. Stallman. *Bison: The YACC-Compatible Parser Generator*, 2002. Available at: http://www.gnu.org/manual/bison-1.25/bison.html.

[3] 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.

[4] V. Paxson. *Flex: A fast scanner generator*, 2002. Available at: http://www.gnu.org/manual/flex-2.5.4/flex.html.

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

[6] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Integrating symbolic worst-case execution time analysis into aspect-oriented software development. OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.

[7] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components. In *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming (WRTP'03)*, Poland, May 2003. Elsevier.

[8] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Towards aspectual component-based real-time systems development. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'03)*. Springer-Verlag, February 2003.