# ASPECT-LEVEL WORST-CASE EXECUTION TIME ANALYSIS OF REAL-TIME SYSTEMS COMPOSITIONED USING ASPECTS AND COMPONENTS [1]

**Aleksandra Tešanović** [*] **Dag Nyström** [**]
**Jörgen Hansson** [*] **Christer Norström** [**]

[*] *Linköping University, Department of Computer Science, Linköping, Sweden, {alete,jorha}@ida.liu.se*
[**] *Mälardalen University, Department of Computer Engineering, Västerås, Sweden, {dag.nystrom,christer.norstrom}@mdh.se*

Abstract: Increasing complexity of real-time systems, and demands for enabling their configurability and tailorability are strong motivations for applying new software engineering principles such as aspect-oriented software development. However, to successfully apply aspect-orientation to real-time systems, methods for designing real-time systems using aspects and components, and analyzing temporal behavior of the resulting software are needed. We contribute by introducing an approach for analyzing the worst-case execution time (WCET) of a real-time system compositioned using aspects and components. The approach consists of aspect-level WCET analysis of components, aspects and the compositioned real-time system, as well as design guidelines for the implementation of components and aspects in a real-time environment. 

Keywords: components, real-time, timing analysis, concurrency control

## 1. INTRODUCTION

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. It follows that development of real-time systems should be based on software technology that supports predictability in the time domain. 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 (Puschner and Burns, 2000).

Increasing complexity in development of real-time systems and the demand for enabling their configurability are strong motivations for adopting new software technologies, such as aspect-oriented software development (AOSD). AOSD allows encapsulating system's crosscutting concerns in "modules", called aspects. AOSD considers components and aspects as two distinct entities where aspects are automatically weaved into functional behavior of components in order to produce the overall system (Lopes *et al.*, 2000).

Applying AOSD in real-time and embedded system development is expected to reduce the complexity of the system design and development, and provide means for a structured and efficient way of handling crosscutting concerns, e.g., synchronization, memory optimization, power consumption, and temporal attributes. However, AOSD does not, in its current form, support predictability in the time domain. Hence, in order to apply

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

AOSD to real-time system development, we need to provide ways of analyzing temporal behavior of aspects, components, and the resulting system.

We contribute by introducing an approach for analyzing WCET of a real-time system compositioned using aspects and components, thus enabling predictable aspect weaving. The approach consists of aspect-level WCET analysis of components, aspects and the compositioned real-time system, and design guidelines for component and aspect implementation in the real-time environment. The aspect-level WCET analysis is based on the concept of symbolic WCET analysis (Bernat and Burns, 2000).

Throughout the paper, to illustrate our approach, we use concrete examples from an on-going research project that is building a configurable embedded real-time database, called COMET (Tešanović *et al.*, 2003). COMET consists of a set of components (e.g., user interface component, transaction scheduler component, and locking component) and a set of aspects (e.g., synchronization, transaction model, security, real-time scheduling, and concurrency control). To show how the implementation of aspects and components supporting predictable aspect weaving should be done, we focus our presentation of one component and one aspect, namely the locking component (LC) and the concurrency control (CC) aspect. Moreover, by implementing CC as an aspect we present a new way of handling concurrency in the real-time database system.

The paper is organized as follows. In section 2 we provide concise background to aspect-oriented software development and symbolic WCET analysis. Design guidelines for aspects and components in real-time systems are presented in section 3. The aspect-level WCET analysis is presented in section 4. The paper finishes with the main conclusions in section 5.

## 2. BACKGROUND

### *2.1 Aspect-Oriented Software Development*

Typically, AOSD implementation of a software system has the following constituents (Kiczales *et al.*, 1997): (i) components, written in a component language, e.g., C, C++, Java, (ii) aspects, written in the corresponding aspect language [2], e.g., AspectC (Coady *et al.*, 2002), AspectC++ (Spinczyk *et al.*, 2002), AspectJ (Team, 2002), (iii) an aspect weaver, which is a special compiler that combines components and aspects.

[2] All existing aspect languages are conceptually very similar to AspectJ, developed for Java.

Components used for system composition in AOSD are not traditional black box components, rather they are gray as we can modify their internal behavior by weaving different aspects in the code of a component [3].

Aspects are commonly considered to be a property of a system that affects its performance or semantics, and that crosscuts the system's functionality (Kiczales *et al.*, 1997). Each aspect declaration consists of advices and pointcuts. A *pointcut* in an aspect language 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.

### *2.2 Symbolic WCET Analysis*

One of the most important elements in real-time system development is temporal analysis of the real-time software. Determining the WCET of the code provides guarantees that the execution time does not exceed the WCET bound. WCET analysis is usually done on two levels (Puschner and Burns, 2000): (i) low-level, analyzing the object code and the effects of hardware-level features, and (ii) high-level, analyzing the source code and characterizing the possible execution paths. Symbolic WCET addresses the problem of obtaining the high-level tight estimation of the WCET by characterizing the context in which code is executed (Bernat and Burns, 2000). Hence, the symbolic WCET technique describes the WCET as a symbolic expression, rather than a fixed constant, in order to provide tighter bounds on the execution time. Typically, the symbolic expression is a function of a variable. For example: for different values of `n` in the loop `for(i = 1; i ≤ n; i + +)` the program has different execution times, so the WCET is a function of `n`, e.g., `WCET(n)=2ms+n*0.1ms`.

## 3. DESIGN GUIDELINES

To enable predictable aspect weaving, i.e., temporal analysis of the real-time aspect-oriented

[3] The internal behavior and attributes of the black box component are strongly encapsulated and cannot be changed or modified.

system, aspects and components should be programmed to conform to the following simple design guidelines. Our experience so far is that these guidelines are not very restrictive.

- Each component provides a set of *mechanisms*, which are basic and fixed parts of the component infrastructure. Mechanisms are fine granule methods or function calls.
- Each component provides a set of *operations* to other components and/or to the system. Operations are represented by coarse granule methods or function calls. Operations are implemented using the underlying mechanisms, which are fixed parts of the component.
- The implementation of the operations determines the initial component policy, as operations implement certain behavior of the component. This initial policy of the component (before any aspects are weaved) we call the *policy framework*, and it is a flexible part of the component, as the implementation of the operations within the framework can change by aspect weaving.
- Aspects can be viewed as policies, since they can change the policy framework of the component by changing one or more operations in the framework.
- The advices in an aspect are implemented using only the mechanisms of the components. Each advice can change the behavior of the component (policy framework) by changing one or more operations in the component. The *around advice* in an aspect can change the behavior of the policy framework by changing the operation implementation (as it is executed instead of the code in the operation). *Before advices* and *after advices* change the behavior of the policy framework without changing the original behavior of the operation (as they are executed before or after the code in the operation).
- The pointcuts in aspects can refer only to the mechanisms, operations in the component, and a type (struct).

Figure 1 shows the LC implemented using these guidelines. The LC assigns locks to requesting transactions and maintains a lock table, thus, it records all locks obtained by transactions in the system. The LC contains the mechanisms (see figure 1) `insertLockRecord()`, `removeLockRecord()`, etc., for maintaining the table of all locks held by transactions. The policy part consists of the operations performed on lock records and transactions holding and/or requesting locks, e.g., `getReadLock()`, `getWriteLock()`, `releaseLock()`. All operations are implemented using underlying mechanisms. The policy framework of the LC, i.e., initial policy of the component, is that lock conflicts are ignored and locks


Fig. 1. The locking component

are granted to all transactions (in this case deadlocks are handled by the application using the LC).

```
aspect CCpolicy{
1: resolveConflict(LC_Operands * op){
2: /*apply specific CC policy to resolve
3:   lock conflict*/
4: }
5:  pointcut getReadLockCall(LC_Operands * op)=
6:    call("void getReadLock(LC_Operands*)")&&args(op);
7:  pointcut getReadWriteCall(LC_Operands * op)=
8:    call("void getWriteLock(LC_Operands)")&&args(op);
9:  advice getReadLockCall(op):
10:   void before(LC_Operands * op){
11:     if the write-lock is already held
12:       then
13:        /*there is a conflict which needs
14:         to be resolved by applying CC policy */
15:        resolveConflict(op);
16:    }
17:  advice getWriteLockCall(op):
18:    void before(LC_Operands * op){
19:      if write- or read-lock is already held
20:       then
21:         /*there is a conflict which needs
22:          to be resolved by applying CC policy */
23:         resolveConflict(op);
24:   }
25: }
```

Fig. 2. The concurrency control aspect

One aspect that influences the policy of the LC, changing its behavior, is the CC aspect, which defines how lock conflicts should be handled in the system. Figure 2 shows the CC aspect `CCpolicy` implementing the pessimistic CC policy HP-2PL (Abbott and Garcia-Molina, 1992), and it has several pointcuts and advices. As defined by the given design guidelines, the pointcuts refer to the operations `getReadLockCall()` and `getWriteLockCall()` (lines 5 and 7). The first pointcut intercepts the call to the function `getReadLock()`, which grants a read lock to the transaction and records it in the lock table. Similarly, the second pointcut intercepts the call to the function that grants a write lock and records it in the lock table. Before granting a read or write lock, the advices (lines 9-16 and 17-24) use the underlying LC's mechanism `findLockRecord` to find the lock record, and then check if there is a lock conflict. If a conflict exists, the advices deal with it by calling the local aspect function `resolveConflict()` (lines 1-9), where the conflict resolution is done by implementing the HP-2PL CC policy. Hence, the `CCpolicy` aspect changes the policy of the LC to HP-2PL.

## 4. ASPECT-LEVEL WCET ANALYSIS

Aspect-level WCET analysis of the real-time system compositioned using components and aspects is based on:

- aspect-level WCET specifications of components and aspects (section 4.1), and
- aspect-level WCET algorithm, which takes the aspect-level WCET specifications as an input and computes the WCET of the components weaved with aspects (section 4.2).

### *4.1 Aspect-Level WCET Specification*

The aspect-level WCET specification of an aspect and a component consists of internal and external WCET specifications. *The internal WCET specification* is a fixed part of the aspect-level WCET specification, and is obtained by symbolic WCET analysis. It represents the WCET of the code not changed by aspect weaving. *The external WCET specification* is a variable part of the aspect-level WCET specification, as it represents the WCET of the code that can be modified by aspect weaving, i.e., which temporal behavior can be changed by "external" influence.

Table 1 gives the relationship between components, aspects, and the aspect-level WCET specification. The temporal behavior of mechanisms, being fixed parts of the component, does not change by aspect weaving. Hence, the WCETs of the mechanisms in the component are determined by the internal WCETs, specified as symbolic expressions. As operations can be modified by aspect weaving, the aspect-level WCET specifications of operations consist both of fixed internal WCET specifications and variable external WCET specifications. External WCETs are specified through usage of mechanisms in operations as aspect weaving changes the operation implementation by changing the number of mechanisms used by the operation. Similarly, advices also have fixed internal WCET specifications and variable external WCET specifications.

Table 1. Aspect-level WCET specifications of aspects and components

| | Internal WCET | External WCET |
|---|---|---|
| Mechanism | x | |
| Operation | x | x |
| Advices | x | x |

Figures 3 and 4 show the aspect-level WCET specification of the LC. For each mechanism, the WCET is determined by symbolic WCET analysis and specified as a symbolic expression (see figure 3). Furthermore, a WCET specification of an operation consists of the internal

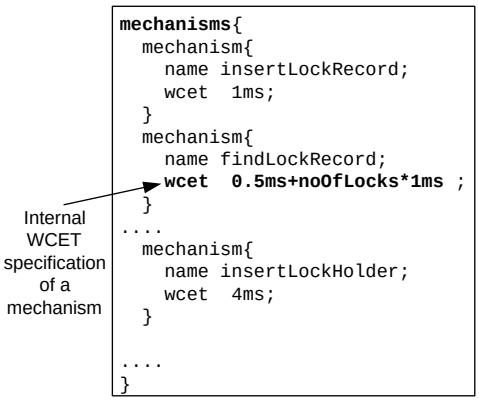


Fig. 3. Aspect-level WCET specifications of the mechanisms of the locking component

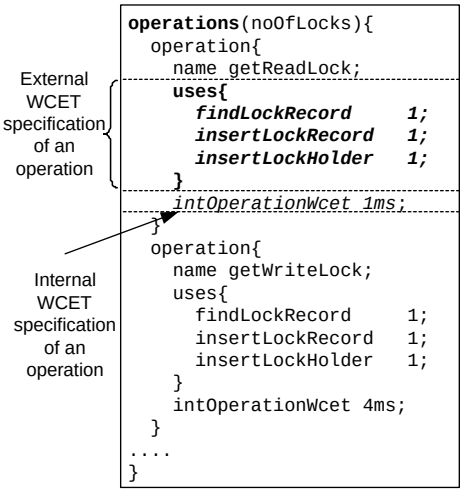


Fig. 4. Aspect-level WCET specifications of the policy framework of the locking component

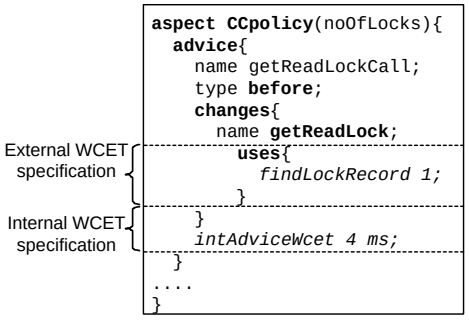


Fig. 5. The aspect-level WCET specification of the `CCpolicy` aspect

WCET specification given as a symbolic expression (`intOperationWcet`), and the external WCET specification expressed through the number of times an operation uses a particular mechanism (see figure 4). As the maximal number of locks in the LC can vary, influencing the execution time of the code, the internal WCETs of operations and mechanisms may be represented as a function of the maximal number of locks in the system, i.e., `noOfLocks`.

The aspect-level WCET specification for the `CCpolicy` aspect, changing the policy framework of the LC, is shown in figure 5.

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


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
```

Fig. 6. The algorithm for aspect-level WCET analysis

The specification includes the internal WCET specification (symbolic expression) of an advice that modified the operation, and the external WCET specification, i.e., the information of the mechanisms used by the advice. For example, the before advice `getReadLockCall` has both an internal and an external WCET specification, which specifies that the advice `getReadLockCall` finds the lock record using `findLockRecord` mechanism of the operation `getReadLock` once, thus influencing the temporal behavior of the real-time system when the `CCpolicy` aspect is weaved into the LC.

### *4.2 Aspect-Level WCET Algorithm*

The algorithm for calculating the WCET of components weaved with aspects, i.e., determining the WCET of the real-time aspect-oriented system, is shown in figure 6. The input data to the algorithm is aspect-level WCET specifications of components and aspects that should be weaved to produce the resulting real-time system. As internal WCET specifications are symbolic expressions, in the initial step of the analysis (before applying the algorithm) all the variables used in symbolic expressions are detected and their values obtained from either a human user or an application.

The algorithm consists of two inter-dependent parts (top-down description):

- `operationWcet()`, which takes the aspect-level WCET specifications of components and aspects and computes the WCET of components weaved with aspects by computing the WCET of all operations in the component modified by aspect weaving, and
- `codeBlockWcet()`, which is called from the `operationWcet()` to compute the WCET of an advice or an operation not weaved with aspects (note, advices and operations use mechanisms as basic blocks).

`operationWcet()` computes the WCET of an operation taking into account that the operation might be modified by aspect weaving. For every advice within an aspect that modifies an operation we need to recalculate the WCET of the operation, depending on the advice type, i.e., whether the advice is type around, before, or after. The WCET of an around advice is calculated directly by a `codeBlockWcet()`, where around advice now is a code block (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 operation since the advice runs before or after the operation (lines 8-11). If the operation for which we are recalculating the WCET calls other operations, then in the WCET of the operation we need to include all the WCETs of every other operation called (which are calculated by the same principle). Thus, we need to have a recursive call to the `operationWcet()` itself (lines 14-17).

`codeBlockWcet()` is used for calculating the WCET of a code block (`codeBlock`), which can be either an advice or an operation. `codeBlockWcet()` does so by first calculating the value of the internal WCET of a given code block based on a symbolic expression (line 1). Then, to obtain an aspect-level WCET of a `codeBlock`, the internal value of the WCET is augmented with the value of the external WCET. The external WCET is computed using the the values of WCET for each mechanism called by the `codeBlock` (lines 3-5), such that the value of WCET of a mechanism (a symbolic expression) is multiplied with the number of times the `codeBlock` uses the mechanism (line 5).

Steps performed in the algorithm are summarized in figure 7. Aspect-level WCET analysis, applied on the LC weaved with the `CCpolicy` aspect, is presented in figure 8. The aspect-level WCET of the LC depends on the WCETs of the operations in the component modified by aspect weaving. The aspect-level WCET of the `getReadLock` operation is calculated by first calculating the WCET of the operation without aspect and then augmenting it for the value of WCET of an advice that modified the operation (see figure 8). As can be seen, for different number of locks in the system this would result in different values of WCET.

(aspectualized)operationWcet = operationWcet (without aspects) + (before/after)adviceWcet
(aspectualized)operationWcet = (around)adviceWcet

operationWcet(without aspects) = codeBlockWcet = internalWcet + externalWcet = intOperationWcet + $\Sigma$ mechanism*usage
adviceWcet = codeBlockWcet = internalWcet + externalWcet = intAdviceWcet + $\Sigma$ mechanism*usage

Fig. 7. Steps performed in the algorithm for obtaining the aspect-level WCET

**(aspectualized)getReadLock**
= (1.1) = getReadLockWcet (without aspects)
+ (before)getReadLockCallWcet
= 11 + noOfLocks*2

where

**getReadLockWcet(without aspects)**
= (2.1) = intOperationWcet + $\Sigma$ mechanism*usage
= 1 + findLockRecordWcet*1 + insertLockRecordWcet*1
+ insertLockHolderWcet*1
= 1 + (0.5+noOfLocks*1)*1 + 1*1 + 1*4 = 6.5 + noOfLocks*1

**adviceWcet**
= (2.2) = intAdviceWcet + $\Sigma$ mechanism*usage
= 4 + 1*findLockRecordWcet
= 4.5 + noOfLocks*1

Fig. 8. Example of applying the aspect-level WCET algorithm to the LC weaved with the `CCpolicy` aspect

## 5. SUMMARY

In this paper we have presented an approach that enables predictable aspect weaving. The approach is called aspect-level WCET analysis and is based on a new algorithm for analyzing the WCET of real-time software composed of aspects and components. The algorithm is based on the symbolic WCET analysis of real-time software systems. While the symbolic WCET technique provides an efficient way for analyzing the WCET of a monolithic (already configured) real-time software, the main goal of aspect-level WCET analysis is determining the WCET 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.

To efficiently analyze aspects when weaved into components, we have given design guidelines for implementing aspects and components in the real-time environment. 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 weaving. As these are initial stages of the work on predictable aspect weaving, there are several open research issues, and we are currently focusing on the following: (i) determining composition rules for combining different configurations of aspect and components, and (ii) developing a tool for aspect-level WCET analysis of the real-time systems compositioned using aspect and components.

## REFERENCES


Abbott, R. K. and H. Garcia-Molina (1992). Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems* **17**(3), 513–560.

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

Coady, Y., G. Kiczales, M. Feeley and G. Smolyn (2002). Using AspectC to improve the modularity of path-specific customization in operating system code. In: *Proceedings of ESEC and FSE-9*.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier and J. Irwin (1997). Aspect-oriented programming. In: *Proceedings of the ECOOP*. Vol. 1241 of *Lecture Notes in Computer Science*. Springer-Verlag. pp. 220–242.

Lopes, C., E. Hilsdale, J. Hugunin, M. Kersten and G. Kiczales (2000). Illustrations of crosscutting. In: *Proceedings of the ECOOP 2000 Workshop on Aspects and Dimensions of Concerns*.

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

Spinczyk, O., A. Gal and W. Schröder-Preikschat (2002). AspectC++: An aspect-oriented extension to C++. In: *Proceedings of the TOOLS Pacific 2002*. Australian Computer Society. Sydney, Australia.

Team,
The AspectJ (2002). The AspectJ programming guide. Xerox Corporation. Available at: http://aspectj.org/doc/dist/progguide/.

Tešanović, A., D. Nyström, J. Hansson and C. Norström (2003). Towards aspectual component-based real-time systems development. In: *Proceedings of the RTCSA 2003*. Springer-Verlag. Taiwan.