Chapter 5

An overview of Mirjam and WeaveC

Authors: István Nagy, Remco van Engelen, Durk van der Ploeg

Abstract In this chapter, we elaborate on the design of an industrial-strength aspect-oriented programming language and weaver for large-scale software development. First, we present an analysis on the requirements of a general purpose aspect-oriented language that can handle crosscutting concerns in ASML software. We also outline a strategy on working with aspects in large-scale software development processes. In our design, we both re-use existing aspect-oriented language abstractions and propose new ones to address the issues that we identified in our analysis. The quality of the code ensured by the realized language and weaver has a positive impact both on maintenance effort and lead-time in the first line software development process. As evidence, we present a short evaluation of the language and weaver as applied today in the software development process of ASML.

5.1 Introduction

One of the primary goals of the Ideals project is to develop methods and tools to improve the handling of crosscutting concerns, such as tracing and profiling. As a solution, a proof-of-concept aspect-oriented [38] language for the C language [64] and weaver (called WeaveC[1]) have been proposed and developed by Durr et al. [33]. Subsequently, a case study has been carried out on a representative component of the ASML software to asses the usability of WeaveC for four particular crosscutting concerns. The outcome of the case-study has shown that these four crosscutting concerns are

[1] Here, the name WeaveC refers both to the AOP language and weaver.

manifested in a significant amount of code (20-40%) besides the code representing the original concerns of the component. By using the CoCoMo model [5], the case-study estimated 5-10% effort reduction and 2-3% lead-time reduction for the code developed with WeaveC, as compared to a non aspect-oriented solution in the C programming language. Although these numbers may seem to be small, it is important to note that all software development (approximately 500 software developers) is affected by the above mentioned and other crosscutting concerns.

Briefly, the case-study demonstrated the industrial maturity of aspect-oriented programming by means of a successful proof-of-concept. As a consequent step, a transfer project was initiated by the industrial partner with the following goals:

- Develop a general purpose aspect-oriented language[2] for the C language that is capable of modularizing crosscutting concerns within ASML software.
- Implement a robust, industrial strength weaver for the proposed aspect-oriented language.
- Develop a way of working (i.e. micro-process) that describes the necessary developer roles, activities and artifacts to deal with crosscutting concerns (by means of the above described programming language and weaver).

The remainder of the chapter is organized as follows: Section 5.2 presents an analysis to identify the requirements of a general purpose AOP language. We also outline a strategy on working with aspects in large-scale software development processes. Section 5.3 discusses the concepts of the proposed aspect-oriented language, weaver and micro-process. Section 5.4 presents the results of an evaluation on the proposed language and weaver. Finally, Section 5.5 draws conclusions.

5.2 Problem analysis

In this section, we present an analysis to identify the requirements towards a general purpose aspect-oriented language to handle crosscutting concerns in the ASML context. The term ‘ASML context’ covers two important design considerations: (a) the existing solution designs (i.e., idioms, see Section 2.2.1) in Chapter 2 of crosscutting concerns and (b) the way of working of software developers within ASML. In the following subsections, we elaborate further on these considerations.

5.2.1 Boundaries of modularization

An important concern of the analysis is to explore the boundaries of modularization of the current solution design of crosscutting concerns. The aim of this step is to determine the set of necessary AOP language abstractions and variation points required

[2]The proof-of-concept AOP language had only a limited set of language abstractions that were necessary for the execution of the case-study.

for the proper modularization. Our objective is not to come up with new language abstractions; in contrast, our objective is to re-use the existing language concepts of aspect-oriented languages as much as possible. For this purpose, we iterate over the essential concepts of a reference model of AOP languages [79, 7]:

```
 1) static ZDSPTI_module_handle ti_handle ;                          //(2)
 2)   ...
 3)   int ZDAPSF_startup ( int sim_mode, boolean caching)
 4)   {
 5)     int result = 0 ;
 6)
 7)     THXAtrace("KI", (1), "ZDAPSF_startup", "> ("                 //(1)
 8)         "sim_mode = %i, caching = %b"                            //(1)
 9)         ")"                                                      //(1)
10)         , mode, caching                                         //(1)
11)       );                                                        //(1)
12)     ...
13)     ZDSPTI_timing_handle timing_hdl = NULL ;                    //(3)
14)     if ( result == OK )                                         //(2)
15)       result = ZDSPTI_register_module("ZDAPSF", &ti_handle);    //(2)
16)     ...
17)     ZDSPTI_timing_in(ZDSPTI_func_timing_hdl, ti_handle,         //(3)
                                 "ZDAPSF_startup", &timing_hdl);  //(3)
18)     ...
19)     if ( result == 0 )
20)        {... }
21)   }
```

Listing 5.1. Illustration of three particular concern instances - Tracing, Setup of Timing and Start of Timing - as they may[a] appear in a particular C function in ASML software. Respectively, these concerns are represented by the lines marked (1), (2) and (3).

[a] The concern instances presented here are slightly altered for reasons of confidentiality. However, this does not affect the essence of the examples.

Base Language Aspect-oriented languages are considered to be extensions of generic programming languages. The fact that ASML software is a legacy system written in the C programming language, restricts the base language to C. Note that the choice of base language will restrict further the design space of other aspect-oriented language abstractions (e.g,. types of join points).

Join points In AOP, (behavioral) join point corresponds to events in the control flow of a program. The example of Listing 5.1 shows that the concern instances typically appear at the beginning of the execution of a function. In other words, the example concern instances need to be executed before (and also after) the occurrence of the

event ‘execution of a function’. Considering the current idioms that realize the crosscutting concerns, we identified two necessary types of join point: *call* and *execution* of a function.

Context of Join points Listing 5.1 shows that various types of context information are required for the modularization of a crosscutting concern. For instance, the concern instance of Tracing refers to the *name* of the function in Line 7, and *formal parameters* of the function in Line 10. The concern instance of Setup of Timing refers to a *local variable* of the function (‘result’) in Line 14. The same concern instance refers to the *name of the module* of the function (‘ZDASP’) in Line 15. Besides these types of context information, we identified that the following types of context information are required: *global parameters* of a function, *types of parameters and local variables, return type of functions* and the fact whether a parameter or local variable behaves as an *input and/or output parameter* in the function (derived information from data-flow analysis).

Context and join points related to pointcuts AOP languages generally make use of the above described context information, not only in the parametrization of crosscutting behavior but also in the designation process of join points (i.e. in pointcut or query languages of AOP languages). Obviously, we will also use these in the designation process of join points.

Context Originated from Build Process In our investigation, we recognized that information derived from the build process is also used in the idioms that realize crosscutting concerns in software. For instance, the concern instance Tracing refers to the component code ("KI") in Line 7; this information is determined from the target of the build process. In a broader view, when crosscutting concerns are woven into different products of a product line, product (and platform) specific information also needs to be addressed in the modularization. Hence, information about product and platform can serve as variation points; these are typically originated from the configuration of the build processes.

Advices and Variation Points in Advices Advices are the units of AOP languages to formulate the crosscutting behavior in terms of the instructions of generic programming languages. All the earlier described types of context information — i.e., the join point, the properties and relationships of a join point, and the platform and product specific information from the build process — can serve as variation points in the formulation of an advice. Normally, these variation points can be *directly* used/referred to through pseudo variables (e.g. `thisJoinPoint` in [65]) and/or through parameters that are extracted from pointcuts and passed to advice bodies (e.g. context exposure in AspectJ).
Besides the direct references and usage, we recognized *indirect* usages of the above listed types of context information. For instance, the string literal ‘`mode = %i,`

`caching = %b`' in Line 8 contains the name of formal parameters and format specifier characters based on the type of the formal parameters. That is, the formal parameters (as join point context) are not explicitly referred to as variations points (unlike in Line 10) but their properties need to be used to *compute* a variation point (cf. the string literal in Line 8) in the notation of the advice-concept.

The fact that we need to deal with a legacy system may have further constraints on the design space of the advice-concept: the crosscutting behavior should be formulated in terms of instructions that can express calls to existing software libraries. For instance, the calls `THXAtrace()` in Lines 7-10 and `ZDSPTI_timing_in ()` in Line 17 are such 'legacy' calls to libraries that realize tracing and timing.

A consequence of this constraint is that the concept of variable-length argument list - denoted by '...' in C - needs to be treated as variation points in the notation of an advice. For instance, in Listing 5.1, the concern instance of tracing is represented by the call `THXAtrace("KI" , (1) , "ZDAPSF_startup" , "> (" "sim_mode = %i, caching = %b" ")", sim_mode, caching)` with 7 arguments, in Lines 7-10. The arguments of this call statement depends the formal parameters `mode` and `caching` of the function `int ZDAPSF_startup ( int sim_mode, boolean caching)` This means that the concern instance of tracing in a different function context will be represented by a call statement with different format string and different number of arguments. For instance, the function `ZDAPSF_shutdown()` without any parameter will have the tracing call `THXAtrace("KI" , (1) , "ZDAPSF_startup" , "> (" ")")` with only 5 arguments. To address this problem, the notation of the advice concept needs to able to deal with the concept variable-length argument list as a variation point. This variable-length argument list can always be derived from the actual join point and its context (cf. the base function to be woven and it formal parameters).

Note that re-implementing the tracing library with a suitable interface is not an option either, as it would introduce other maintenance problems related to legacy systems.

Aspectual States/Aspect Instances In Listing 5.1, both the concern instances of Setup of Timing and Start of Timing use the variable `ti_handle` that represents the module handler of Timing. The concern instance of Timing uses another variable, called `timing_hdl` in Line 17. The difference between these two variables is apparent from the example already: `ti_handle` is declared as a global static variable in Line 1. This means that `ti_handle` can (and will) be used in every function of the module; thus, it can be *shared* among different concern instances that appear in different functions. In contrast, the variable `timing_hdl` is declared as a local variable of the function in Line 13; hence it is *local* only to those concern instances that appear only within the same functions. In terms of aspect-oriented programming, we identified the need of two types of aspectual states: one which is shared only among those advices that are woven at the same join point (*per join point*) and one which is globally shared among every advice (i.e., *per aspect*).

Note that there are many other fundamental concepts, e.g. ordering of advices, context exposure, et cetera, in the design space of aspect-oriented languages. Naturally,

we consider these concepts as parts of our design space; however, the discussion on the motivation for each particular language concept is beyond the scope of this chapter.

5.2.2 Quality aspects of concrete syntax for large-scale development

In the previous section, we discussed the boundaries of modularization that determines the necessary expressiveness of the language - i.e., abstract-syntax of a language, in terminology of Domain Specific Languages [42]. However, besides necessary expressiveness, there are various quality aspects on the concrete syntax that are significant from the point of view of large-scale software development. In this section, we discuss these.

Predictability (in design phase) Predictability ensures developers that certain properties are held during the development of software. This is crucial in large-scale development to prevent mistakes and errors already in the design phase as soon as possible. Besides, when introducing new technology into a large organization, ensuring predictability in the design phase is well-motivated for the following two reasons: (1) it can lessen the risk of improper use (and its undesired side effect) of the new technology; (2) it can reduce the fear of using a new technology among the developers (e.g., by providing well-controlled means for the new technology).

To this aim, we are going to use well-known language mechanisms, such as type-checking, enforcement of declarative completeness, and an extensive set of syntactic and semantic rules in the design of the aspect-oriented language[3].

Evolvability Evolvability is an important software quality factor that indicates the ability of developing programs incrementally. In general, evolvability facilitates extending an application towards new requirements mostly by reusing previously written modules with minimal or no modification. By ensuring evolvable specification, we expect to reduce the complexity of the code caused by the frequent appearance of the phenomenon called ‘deviations from standard functionality’ in a large-scale development.

To this aim, we are going to provide language abstractions that (1) can be re-used in different specifications and (2) can support the re-use of existing specifications.

Extendibility In principle, language constructs that provide means of parametrization can positively contribute to the extendibility of a language. Providing ease of extendibility for a language is beneficial in large-scale development for two reasons. First, due to time-to-market pressure the language (and also the weaver) needs to be incrementally delivered. Secondly, rolling out a new version of a language and weaver is not a trivial task as it affects many ongoing software development activities; hence, a

[3]These language mechanisms obviously require adequate compiler and/or run-time support.

large number of development modules maintain dependencies to aspect specifications. This means that change requests towards the language should result in changes in the notation as *minimal* and *isolated* as possible.

Comprehensibility We define comprehensibility as the ability to understand the meaning of a program by just looking at its source code. Comprehensibility can be influenced by the programming style and the notation of the abstractions of the adopted language, such as how the program units are modularized, where the references in the units are specified, and the style of notation that reflect the underlying computational paradigm. Although this quality aspect had less significance compared to the previous ones in our list, we needed to take into account the fact that developers at ASML have a stronger background on procedural and object-oriented languages (with strong typing) compared to logic or functional languages.

5.2.3 Expected way of working

In the previous two sections, we discussed the requirements towards an aspect specification language used in large-scale software development. However, when introducing aspect-oriented programming (or any other new technology) into a large organization, a clear and sound strategy on the expected way of working is also necessary, besides the required means and artifacts. In this section, we discuss our strategy on how aspects are intended to be used in the software development process of ASML.

To make sure that developers can benefit from the advantages[4] of aspect-oriented programming, the objective is that *every* developer should be able to *use* aspects easily with a minimal learning curve. On the other hand, to minimize the possible danger of the improper use[5] of AOP, only *a few* of the developers are allowed to *write* and *release* news aspects. To this aim, our objective is to provide generic aspects with highly and easily customizable interfaces:

- Developers are expected to use aspects in a standard way (seamlessly, by enabling them in the build process).
- Most users will use aspects with their standard functionality.
- Some users will want to deviate from the standard functionality. In other words, they want to customize the functionality of generic aspects according to their special needs. For instance, some users will want to ‘switch off’ the tracing of certain time-critical functions.
- This customization should be minimal and rely on design information added to the source code, as most users are not allowed to modify aspects or write their own one.

[4] E.g., locality of changes, consistency and clarity of code, et cetera.
[5] E.g., undesired side-effects in the control-flow, ‘patching by aspects’, et cetera.

5.3 Mirjam, an aspect-oriented language extension for C

In this section, we outline the important constructs and characteristic properties of the realized aspect-oriented language, called Mirjam. Due to a lack of space, we cannot iterate over the full design space of Mirjam. The interested readers can find a more detailed introduction and description of the language in [9].

5.3.1 Aspect

The main unit of modularization is the aspect specification file. An aspect specification may contain two language constructs (see Listing 5.2): *context declaration closure* and *aspect declaration*.

```
 1)   context{
 2)       #include "THXAtrace.h"
 3)       #define FALSE 0
 4)
 5)       typedef int boolean;
 6)   }
 7)
 8)   aspect SimpleAspect
 9)   {
10)       advice someAdvice() before (FunctionJP JP)
11)       {
12)         boolean tracing_flag = FALSE;
13)         THXAtrace(JP.module.name, JP.name, tracing_flag, "> ");
14)      }
15)     ...
16)   }
```

Listing 5.2. Illustration of the context declaration closure and aspect declaration in an aspect specification file.

The context declaration closure is a placeholder for a standard C declaration. We will use the declarations in C in the context closure to ensure *declarative completeness* in the notation of advices. The context closure in the listing above has two preprocessor directives in Lines 2 and 3, and a typedef declaration in Line 5. To resolve the preprocessor directives, the aspect specification is first preprocessed by a standard C preprocessor. The declarations in the context closure are used later in the specification of the crosscutting behavior (i.e., advice in term of Mirjam) in Lines 11 and 12. Note that the function call 'THXAtrace' (in Line 13) is declared in the included THXAtrace.h file.

The aspect declaration part is a container type of program element: it works as a name space. The contained language abstractions (query, advice- and binding-declarations) can be referred to through this name space.

5.3.2 Queries as pointcuts

The language concept of pointcut is realized by the language abstraction *query* in Mirjam. A query in Mirjam returns a set of join points. A join point is a *tuple* that contains an arbitrary number of (but at least one) tuple elements. A tuple element can be of two types: *join point location* and *join point context*. As the names suggest, the type of join point location describes the place where we want to weave in crosscutting behavior (e.g., execution of a function), while the type of join point context describes the context of the weaving. We can use the context information for (1) either refining of the weaving location or (2) customizing the crosscutting behavior to be woven to certain weave contexts.

```
1)   int a;
2)   int f(int b, int c) {
3)      ...;
4)   }
5)   int g(double d) {
6)
7)   }
```

Listing 5.3. An example of base code that we will use in the rest of the discussion to illustrate how the query language of Mirjam works.

We will use a number of examples of queries to present the characteristic features of the query language of Mirjam:

```
1)   query Q1() provides (FunctionJP JP)
2)   {
3)      JP: true;
4)   }
5)
6)   query Q2() provides (FunctionJP JP)
7)   {
8)         JP: JP.name =~ "f.*";
9)   }
10)
11)  query Q3() provides (FunctionJP JP, Variable@JP V)
12}  {
13)     JP: true;
14)     V : true;
15)  }
16)
17)  query Q4() provides (FunctionJP JP, Variable@JP int V)
18)  {
19)     JP: tuple(JP) in Q1() && |JP.formalParameters()| >= 1;
20)     V : V in JP.formalParameters();
21)  }
```

```
22)
23)   query Q5() provides (FunctionJP JP, Variable@JP[] V)
24)   {
25)      JP: tuple(JP) in Q1() && |JP.formalParameters()| >= 1;
26)      V : V == JP.formalParameters() ;
27)   }
```

Listing 5.4. Examples of query-declarations in Mirjam.

As an example of a query, consider the query declared between Lines 1 and 4 in Listing 5.4. The query declaration starts with the keyword `query` followed by an identifier ('Q1') and a list of possible formal parameters. After the formal parameters we define a set of tuple variables preceded by the keyword `provides`. The list of tuple variables describes the elements and type of tuples that the given query provides. In Lines 1-4, the query Q1 provides tuples with one tuple element, the type of this tuple element is FunctionJP. The type FunctionJP represents the type of the join point location of the execution of a function. Inside the query, a tuple condition needs to be defined for each tuple variable. In Line 3, the tuple condition 'true' means that every particular function execution will satisfy this query; i.e., there is no further restriction on the tuple variable JP. The resulting set of tuples of query Q1 executed on the base code of Listing 5.3 is {(f),(g)}[6]. In Lines 6-9, the query Q2 is similar to Q1 except that it has a more restrictive tuple condition in Line 8: the name of the executing function shall start with the prefix 'f' (defined by the regular expression `'f.*'`). Similar to the notation of object-oriented programs, the dot operator can be used to access instance variables and execute methods of certain types in Mirjam. The result set of Q2 on the given base code is (f), as there is only one function that can satisfy this condition.

In Lines 11-15, the query Q3 will provide set of tuples with two tuple elements. The first tuple variable is of type 'function execution'. The second, new tuple variable is of type Variable, which is a type of join point context. More precisely, the type Variable can represent global variables, formal parameters and local variables of an executing function. In Line 11, the declaration Variable@JP means that we expect the variables in the scope of the function execution of JP; that is, there is direct link between a particular function and variable in the result set of the query. Note that a second tuple condition 'true' has been declared for the second tuple variable in Line 14. When a query is evaluated, the query engine iterates over the possible values of both tuple variable types. During the iteration, if there is a combination of particular values of tuple variables that can satisfy both tuple conditions (based on the available base code), a tuple is created and added to the result set of query. In short, the query Q3 returns an ordered set of tuples (i.e. the Descartes-product) of all functions and variables related with those functions: {(f,a), (f,b), (f,c), (g,a),(g,d)} based on the code of Listing 5.3.

In Lines 17-21, the query Q4 illustrates a bit more complex use of queries. The tuple variable V in Line 17 is declared with a type restriction: the C type of the variable

[6]For the sake of simplicity, the letters f and g represent the execution of functions f and g.

is restricted to int. In Line 19, we show a possible re-use of a previously declared query: the tuple variable JP is converted to a tuple by the conversion operator 'tuple'. The membership relation 'in' specifies that JP (as converted to a tuple of one element) should be in the result set of the query Q1. The membership relation ('in') is not the only way to re-use queries; the equivalence relations can also be used with query calls. Queries can be called with formal parameters to perform selection or projection (in terms of tuple relational calculus [24])[7] on previously defined queries. In Line 19, the second part of the tuple condition specifies that the executing function needs to have at least one formal parameter. The second tuple condition in Line 20 specifies that V is a formal parameter of JP. The execution of Q4 on the provided base code of Listing 5.3 results in the following set of tuples: {(f,b), (f,c)}.

In Lines 23-27, the query Q5 is slightly modified version of the query Q4. Q5 differs from Q4 in two places. First, the type of the tuple variable V is an array type denoted by the symbols '[ ]' in Line 23. Note that this tuple variable has no C type specification either. Secondly, the tuple condition uses the equivalence relationship ('==') instead of the membership relationship ('in'). This means that Q5 will provide tuples with two elements: a tuple element of a function-execution and a tuple element of an array of variables. The equivalence relationship indicates that the array of variables must be the array of formal parameters of the corresponding function. This means the result set of Q5 is { (f, {b,c}), (g,{d}) } based on the code of Listing 5.3.

5.3.3 Advices

Similar to other AOP languages, the program unit that specifies the crosscutting behavior is called *advice* in Mirjam. The crosscutting behavior is defined in terms of instructions of the C programming language.

```
1)   advice printIntParam(Variable@JP int V) before (FunctionJP JP)
2)   {
3)    printf("In module %s, function %s executes with argument %s=%d",
4)               JP.module.name, JP.name, V.name, V);
5)   }
```

Listing 5.5. An example of advice-declaration.

As an example of an advice, consider the advice declared in Listing 5.5. The declaration starts with the keyword 'advice' and is followed by an identifier and a (possibly empty) list of formal parameters. A formal parameter has to have a type specifier in Mirjam and may have a type specifier in C. The formal parameters are followed by the type of the advice that specifies whether the advice is to be executed before or after the execution of a join point. Finally, the last element in the signature of the advice is the actual join point variable in the form of a formal parameter (FunctionJP JP). In the

[7] Defining the formal semantics of queries is beyond the scope of this chapter.

body of the advice, we refer to the properties of the join point through this variable in Line 4. In the same line, we also use the parameter V as a formal parameter similar to the formal parameters used in C.

Note that the correct use of the format specifier (e.g., %d in Line 3) cannot be checked just like in normal C code. For instance, we could use %f that would have resulted in an incorrect execution. This piece of advice exemplifies another possible problem as well: if we coupled it with the query Q4 (of Listing 5.4) the advice would be woven two times on the function `f`, as Q4 returns {(f,b), (f,c)}. This means that the execution of the program would have two output messages, see Listing 5.6.

```
6)  In module ZZ, function f executes with argument b=5
7)  In module ZZ, function f executes with argument c=11
```

Listing 5.6. Illustration of output messages from advice executions.

This execution is correct because Q4 exposes only formal parameters of type int. However, if the query is not restrictive enough, formal parameters with other types than int (e.g. double, pointer-type, et cetera) can be queried too. This would result in incorrect output messages, since the advice expects a parameter of type int[8]. In addition, having the same message two times, with only a small difference at the end, is not considered a neat solution: a single printf statement could handle a variable number of arguments. So the question is how to specify the statement printf with variable number of arguments of different types that depend on the weaving context? Note that we discussed the same problem in Section 5.2.1 at 'Variation Points in Advices'.

5.3.4 Advice generators

To handle the category of problems described above, we introduced the language construct *advice generator*. These are special built-in constructions that can be used in the body of an advice and are recognizable by the form `@type<...>@`. The first element of a generator is usually an iterator of the form `[iterator: expression]`. The iterator contains the name of an array variable and the (local) name for each element, and returns a list of expressions. The `expression` will be evaluated for each element. Finally, the `type` says how the expressions are combined and what the outcome of the advice generator is[9].

[8] To prevent this sort of errors, we do type checking between the type of the actual tuple values and the type of the advice-parameter.

[9] In other (typically, functional) languages one can achieve the same functionality by using the list manipulator functions *map* and *join* on lists.

```
 1)   advice printParams(Variable@JP[] Args) before (FunctionJP JP)
 2)   {
 3)     printf("Function %s executes with argument(s) "
 4)       @StringConstant<
 5)       [arg in Args:
 6)             strConcat(arg.name, "=",formatString(arg))],
 7)        "; " //semicolon as separator
 8)       >@
 9)       ,JP.name
10)       ,@VarargExpression<[arg in Args: formatExpression(arg)]>@
11)     );
12)   }
```

Listing 5.7. An example application of two advice generators.

For an example application of advice generators, consider Listing 5.7. In Lines 4-8, the advice generator `@StringConstant` will generate a format string for an array of variables passed to the advice (Args)[10]. In Line 6, the built-in function `strConcat` creates a piece of string from the name of a variable (cf. formal parameter of a function), from the equation sign ('=') and from the result of the function `formatString` returning the format specifier of the C type of the variable. In Line 5, this concatenation function is applied on each variable in the array of the variables ('arg in Args'). This will result in a list of a piece of format string for each formal parameter. The resulting list of format strings per parameter will be combined into a final string literal by the StringConstant generator, using the semicolon as a separator. Similarly, a variable-length argument list will be created by the advice generator `@VarargExpression` in Line 10. If we couple this advice (Listing 5.7) with the query Q5 (in Listing 5.4) the advice is woven once on the functions f and g, as Q5 returns (f, {b,c}), (g,{d}). This means that the execution of the program would have two output messages, see Listing 5.8.

```
1)   Function f executes with argument(s) b=5; c=11
2)   Function g executes with argument(s) d=3.1415
```

Listing 5.8. Illustration of output messages from advice executions with advice generators.

5.3.5 Bindings

The third language construct that can be declared within an aspect is the *binding closure*. The binding closure contains typically one or more *binding definitions* introduced

[10]This list of parameters is derived from the context of join point: they are e.g., representing formal parameters of a function. In the following section, we will show both the query that realizes this derivation and the binding-definition that couples the advice printParams with that query.

by the keyword `foreach`. One binding definition can couple a query with one or more *binding entities*. Different types of binding entities exist in Mirjam: for example, the binding entity `apply` binds advices to queries, and passes the result of the queries to the advices using a parameter-passing mechanism. When more than one advices are to be applied to the same join point, one can specify the order of their application by the binding entity `order` per join point. The binding entity `error` and `warn` can raise errors, when certain situations occur in the base code. Finally, the binding closure and binding entities may also contain variable declarations, called binding variables, to share states between advices. In the following subsections, we will show the use of each of these language constructs.

```
 1)   aspect VerySimpleTracing
 2)   {
 3)     query Q5() provides (FunctionJP JP, Variable@JP[] V)
 4)     {...}
 5)
 6)     advice printParams(Variable@JP[] Args)before(FunctionJP JP)
 7)     {...}
 8)
 9)     binding
10)     {
11)       foreach (thisFunc, vars) in Q5
12)       {
13)           apply on thisFunc {
14)             printParams(vars);
15)           }
16)       }
17)     }
18)   }
```

Listing 5.9. An example of binding and an illustration of passing context information to advices.

For an example of a binding-declaration, consider Listing 5.9. In Line 3, the query Q5 provides a set of tuples of a join point variable (FunctionJP JP) and a context variable (Variable@JP V). In Line 11, the binding definition iterates over the result set of Q5: `thisFunc` and `vars` will represent the two variables of each tuple. For each iteration step, these variables hold the values of the actual tuple elements. In Lines 13 and 14, the actual values of `thisFunc` and `vars` are passed as parameters to the advice in the advice-call. Finally, in Line 6, the variables are ‘received’ and they are used in the body of the advice. This type of parameter passing works in a similar way to the macro substitution mechanism of the C preprocessor.

5.3.6 Bindings variables

As we mentioned in the previous section, the binding closure and binding entities may also contain variable declarations, called *binding variables*, to share states between the executions of advices.

```
 1)   binding {
 2)      static int ti_handle;
 3)
 4)      foreach (startup) in startupFunction()
 5)      {
 6)         apply on startup {
 7)            moduleRegistration(ti_handle);
 8)         }
 9)      }
10)
11)      foreach (f) in timedExecution()
12)      {
13)        apply on f {
14)           int timer;
15)
16)           functionTimingStart(ti_handle, timer);
17)           functionTimingStop(timer);
18)        }
19)      }
20)   }
```

Listing 5.10. Illustration of the use of binding variables.

Consider the variables `static int ti_handle` and `int timer` in the Lines 2 and 14 of Listing 5.10. One can use binding variables *per aspect* (like `ti_handle` in Line 2) and *per join point* (like `timer` in Line 13). 'Per aspect' binding variables, e.g., `ti_handle`, are shared between all advices within a binding. This means that if one advice changes `ti_handle`, the change can be observed in all other advices of the aspect. 'Per join point' binding variables, e.g. `timer`, are shared only among those advices that are called from the binding entity where the variable is declared. For example, `timer` is shared only between the execution of the advices `functionTimingStart` and `functionTimingStop` in each function where they are applied together.

5.3.7 Annotations

Annotations in WeaveC are used to attach semantic meaning to syntactical constructs in a program written in the programming language C. The annotation mechanism is designed such that the following objectives are met:

Easy of use An application of an annotation to a program should have minimal overhead, both in terms of the effort spent by a developer and of the impact on the program listing. The syntax should be unobtrusive (not disturb the natural flow of the program listing) and similar to the syntax of similar C concepts (type modifiers like storage class, const et cetera).

Flexible for different usage The application of annotations is only limited by the scope of tools that can process them, so the mechanism should place few restrictions and allow for varied use.

Allow checking of the correct application of annotations This means that it must be detectable by a tool if an annotation is applied at the wrong location (i.e., to a function while it makes no sense for a function), with the wrong arguments (i.e., with a value for a specific field while the annotation has no such field) or multiple times (when it makes no sense to do so for the annotation). It does not need to be possible to check for unexpected absence of annotations (i.e., a function has no annotation while one is always expected) or illegal combinations of annotations or values of fields of annotations (i.e. a function can not have both annotations 'a' and 'not_a'). These checks, if desired, must be performed by the tools that make use of the annotations.

Easy to hide from a program This should be the case for instance if the program is to be read by tools that are unaware of this annotation mechanism. Such hiding should be possible by a standard-compliant C pre-processor or a standard text processing facility.

Robust Common typing errors in the declaration or application of an annotation should not lead to undesirable parsing of the actual program code.

5.3.8 An example of using annotations

In this section, we outline of how the annotation mechanism of WeaveC can be used.

```
 1)   // --- an annotation-declaration in .c files ---
 2)   /*$
 3)      annotation{
 4)        boolean value;
 5)      } trace[module,function,parameter,variable,type] = {
 6)        value = TRUE
 7)      };
 8)   $*/
 9)
10)   // --- an annotation-application in .c files ---
11)   int critical_function()
12)   /*$trace(FALSE)$*/
13)   {...}
```

```
14)
15)   // --- a query referring to an annotation-application  ---
16)   query allTraceableFuncs() provides (FunctionJP JP)
17)   {
18)        JP:  (JP.name != "main") &&
19)             (!( (JP.$trace?)&&(JP.$trace.value == false) ))
20)   }
```

Listing 5.11. Illustration of the declaration, application and use of annotations.

The declaration and application of annotations are denoted by the tokens `/*$` and `$*/`, in the style of Splint [40]. In Listing 5.11, the annotation `trace` is declared between Lines 2-8. The annotation declaration has one boolean field called `value` (in Line 4) and it can be applied on modules, functions, formal parameters, variable declarations and type declarations, as indicated in Line 5. In Line 6, we specify that the default value for the field `value` is TRUE[11]. An application of the annotation `trace` is illustrated in Line 12: the application is part of the signature of the function declaration `int critical_function()`. Finally, the query `allFunctions()` shows a use of annotations for determining join points in Mirjam, in Lines 18-19. Only those function executions will be designated that do not have the name "main" and do not have the application of the annotation 'trace' with the value FALSE.

5.4 Evaluation

Dürr et al. [34] carried out an experiment on quantifying the benefits of using aspect-oriented programming by means of the above defined language and weaver. The setup of the complete experiment and all statistical data about the results can be found in [34]. In this section, we provide only a summary about the setup and the results of the experiment.

The goal of the experiment was to determine whether using AOP speeds up the development and maintenance of the ASML codebase. The experiment consisted of five change scenarios in two sets related to use of the concern Tracing. Twenty software developers participated in the experiment[12]; the participants were split into two groups. The most important reason to do the splitting was to verify that the two sets of change scenarios were equivalent.

Although it is hard to extract significant results from the experiment due to small number of participants, the following conclusions can be definitely supported from the results of the experiment:

- Adding tracing to a function takes considerably less time with AOP than without.

[11] This is a standard predefined macro. The weaver has a built-in C preprocessor to resolve macro substitutions within annotation declarations and applications.

[12] The experiment was actually combined with training on the use of aspects and WeaveC.

- Removing tracing from a function takes more time with AOP than without. This is probably caused by the (first) usage of annotations.
- Selectively tracing parameters in a function, takes less time with AOP than without.
- Adding tracing to a function manually introduces significantly more errors than with AOP.
- Changing the signature of a function manually introduces significantly more errors than with AOP.

Besides this experiment, we are continuously working on the evaluation of the language and weaver by different means. At the moment of writing, software developers and engineers (approximately 70-80 participants) - who are currently using the language and weaver - are participating in a survey to evaluate the tool in the view of usability and other quality concerns. We consider the feedback both from the experiment and survey crucial in driving our design on the upcoming features of both the language and the weaver.

5.5 Conclusions

Currently, WeaveC is part of the standard build process of the ASML software. WeaveC is used in 42 components today; in these components 298 targets are generated based on 1007 woven source files. Software developers do not need to write tracing and timing code anymore. The aspect files that realize these crosscutting concerns are part of the external interface of a standard software component. The weaving of these aspects is enabled by make-files [95] in the build process. Besides, developers can add annotations to their base code to customize the standard tracing functionality when needed.

Mirjam and WeaveC can fulfill the promises of the proof-of-concept weaver. The quality of the code ensured by Mirjam and WeaveC has positive impacts both on the maintenance effort and lead-time in the first line software development process. The increased quality of the code also improves lead-time and reduces errors in terms of the analysis of problems/machine performance, especially during integration and field problems. As a result, the forth line software development can also be done faster and better, which therefore reduces integration time and improves the response to field problems.

References

[1] Adams E.N. Optimizing Preventive Service of Software Products. *IBM Journal of Research and Development*, volume 28(1):pp. 2–14, 1984.

[2] Bekkering T., Glas H., and Klaassen D. *Management van processen - Succesvol realiseren van complexe initiatieven*. Het Spectrum, 2004.

[3] Bergmans L. and Akşit M. Principles and Design Rationale of Composition Filters. In R.E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pp. 63–95. Addison-Wesley, Boston, 2005.

[4] Bernstein A.J. Program Analysis for Parallel Processing. *IEEE Trans. on Electronic Computers*, volume EC-15:pp. 757–762, October 1966.

[5] Boehm B.W. *Software Engineering Economics*. Prentice-Hall, 1981.

[6] van den Brand M., van Deursen A., Heering J., de Jong H.A., de Jonge M., Kuipers T., Klint P., Moonen L., Olivier P.A., Scheerder J., Vinju J.J., Visser E., and Visser J. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pp. 365–370. Springer, 2001.

[7] Brichau J. and Haupt M. Survey of Aspect-Oriented Languages and Execution Models.

[8] Brodie M.L. and Stonebraker M. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann, 1995.

[9] Brouwer S. and Nagy I. Mirjam. Internal document 107731, ASML, 2007.

[10] Bruntink M. Aspect Mining Using Clone Class Metrics. In *Proceedings of the 2004 Workshop on Aspect Reverse Engineering (co-located with the 11th Working Conference on Reverse Engineering (WCRE'04))*. November 2004. Published as CWI technical report SEN-E0502, February 2005.

[11] Bruntink M. Linking Analysis and Transformations Tools with Source-based Mappings. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pp. 107–116. IEEE Computer Society Press, September 2006.

[12] Bruntink M. Analysis and transformation of idiomatic crosscutting concerns in legacy software systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pp. 499–500. IEEE Computer Society Press, 2007.

[13] Bruntink M., van Deursen A., D'Hondt M., and Tourwé T. Simple Crosscutting Concerns Are Not So Simple – Analysing Variability in Large-scale Idioms-based Implementations. In *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development (AOSD'07)*, pp. 199–211. ACM Press, March 2007.

[14] Bruntink M., van Deursen A., van Engelen R., and Tourwé T. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 200–209. IEEE Computer Society, 2004.

[15] Bruntink M., van Deursen A., van Engelen R., and Tourwé T. On the Use of Clone Detection for Identifying Cross Cutting Concern Code. *IEEE Transactions on Software Engineering*, volume 31(10):pp. 804–818, 2005.

[16] Bruntink M., van Deursen A., and Tourwé T. An initial experiment in reverse engineering aspects from existing applications. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, pp. 306–307. IEEE Computer Society, 2004.

[17] Bruntink M., van Deursen A., and Tourwé T. Isolating Idiomatic Crosscutting Concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pp. 37–46. IEEE Computer Society, 2005.

[18] Bruntink M., van Deursen A., and Tourwé T. Discovering Faults in Idiom-Based Exception Handling. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*, pp. 242–251. ACM Press, 2006.

[19] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley series in Software design patterns. John Wiley & Sons, 1996.

[20] Bush M. Improving software quality: the use of formal inspections at the JPL. In *Proceedings of the International Conference on Software Engineering*, pp. 196–199. IEEE Computer Society, 1990.

[21] Christian F. *Exception handling and tolerance of software faults*, chapter 4, pp. 81–107. John Wiley & Sons, 1995.

[22] Clarke E.M., Grumberg O., and a. Peled D. *Model Checking*. The MIT Press, 1999.

[23] Coady Y., Kiczales G., Feeley M., and Smolyn G. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC'01) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'01)*, pp. 88–98. ACM Press, June 2001.

[24] Codd E. A Relational Model of Data for Large Shared Data Banks. volume 13, pp. 377–387. 1970.

[25] Colyer A. and Clement A. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pp. 56–65. ACM Press, New York, NY, USA, 2004. doi:http://doi.acm.org/10.1145/976270.976279.

[26] Coplien J. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1991.

[27] Csertán G., Huszerl G., Majzik I., Pap Z., Pataricza A., and Varró D. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In *ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pp. 267–270. IEEE Press, 2002.

[28] Demeyer S., Ducasse S., and Nierstrasz O. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[29] Durr P., Bergmans L., and Aksit M. Technical Report: Formal model for SECRET. Technical report, University of Twente, 2005.

[30] Durr P., Bergmans L., and Aksit M. Reasoning about Semantic Conflicts between Aspects. In R. Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pp. 10–18. Lancaster University, July 2006.

[31] Durr P., Gulesir G., Bergmans L., Aksit M., and van Engelen R. Applying AOP in an Industrial Context. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*. March 2006.

[32] Durr P., Staijen T., Bergmans L., and Aksit M. Reasoning about Semantic Conflicts between Aspects. In *EIWAS '05: The 2nd European Interactive Workshop on Aspects in Software*. Brussel, Belgium, September 2005.

[33] Durr P.E.A. and Bergmans L.M.J. High-level Design of WeaveC. Internal document, ASML, 2006.

[34] Durr P.E.A., Bergmans L.M.J., and Aksit M. Initial Results for Quantifying AOP. Technical Report TR-CTIT-07-71, Centre for Telematics and Information Technology, University of Twente, 2007.

[35] Durr P.E.A., Bergmans L.M.J., and Aksit M. Static and Dynamic Detection of Behavioral Conflicts between Aspects. In *Proceedings of the 7th Workshop on Runtime Verification*, number 4839 in LNCS, pp. 38–50. Springer Verlag, Vancouver, Canada, March 2007.

[36] Dyer M. The Cleanroom Approach to Quality Software Development. In *Proceedings of the 18th International Computer Measurement Group Conference*, pp. 1201–1212. Computer Measurement Group, 1992.

[37] Elrad T., Filman R.E., and Bader A. Aspect-Oriented Programming. *Comm. ACM*, volume 44(10):pp. 29–32, October 2001.

[38] Elrad T., Filman R.E., and Bader A. Aspect-Oriented Programming: Introduction. *Communications of the ACM*, volume 44(10):pp. 29–32, October 2001.

[39] Engler D.R., Chelf B., Chou A., and Hallem S. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *4th Symposium on Operating System Design and Implementation*, pp. 1–16. USENIX Association, 2000.

[40] Evans D. and Larochelle D. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, Jan/Feb 2002.

[41] Fenton N.E. and Pfleeger S.L. *Software Metrics: A rigorous and Practical Approach*. PWS Publishing Company, second edition, 1997.

[42] Fowler M. Language Workbenches: The Killer-App for Domain Specific Languages? Technical report, June 2005.

[43] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns*. Addison-Wesley, 1995.

[44] Ganter B. and Wille R. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[45] Gool L., Punter T., Hamilton M., and Engelen R. Compositional MDA. In Nierstrasz, O. et al, editor, *Proceedings of ACM/IEEE Models 2006*, volume 4199 of *Lecture Notes in Computer Science*, pp. 126–139. Springer, 2006.

[46] Graaf B. *Model-Driven Evolution of Software Architectures*. Ph.D. thesis, Delft University of Technology, November 2007.

[47] Graaf B., Lormans M., and Toetenel H. Embedded Software Engineering: The State of the Practice. *IEEE Software*, volume 20(6):pp. 61–69, November–December 2003.

[48] Graaf B., Weber S., and van Deursen A. Migration of supervisory machine control architectures. In R. Nord, N. Medvidovic, R. Krikhaar, J. Stafford, and J. Bosch, editors, *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pp. 261–262. IEEE CS, November 2005.

[49] Graaf B., Weber S., and van Deursen A. Migrating supervisory control architectures using model transformations. In G. Visaggio, G. Antonio Di Lucca, and N. Gold, editors, *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pp. 151–160. IEEE Computer Society, 2006.

[50] Graaf B., Weber S., and van Deursen A. Model-driven migration of supervisory machine control architectures. *Journal of Systems and Software*, 2008. Doi: 10.1016/j.jss.2007.06.007.

[51] Gulesir G., Bergmans L., Durr P., and Nagy I. Separating and managing dependent concerns. LATE 2005 workshop at AOSD 2005.

[52] Gurzhiy T. Model Transformations using QVT - Feasibility Analysis by Implementation. SAI Technical Report 2006056, Eindhoven University of Technology, August 2006.

[53] Hannemann J., Chitchyan R., and Rashid A. Analysis of Aspect-Oriented Software, Workshop report. In *ECOOP 2003 Workshop Reader*. Darmstadt, Germany, July 2003.

[54] Hardebolle C., Boulanger F., Marcadet D., and Vidal-Naquet G. A generic execution framework for models of computation. In *Model-Based Methodologies for Pervasive and Embedded Software*, pp. 45–54. 2007.

[55] Hooman J. and van der Zwaag M.B. A semantics of communicating reactive objects with timing. *International Journal on Software Tools for Technology Transfer*, volume 8(2):pp. 97–112, 2006.

[56] Huang J. and Voeten J. Predictable model-driven design for real-time embedded systems. In *Proceedings of Bits & Chips conference.* 2007.

[57] Huang J., Voeten J., and Corporaal H. Predictable real-time software synthesis. *Real-Time Systems Journal*, volume 36(3):pp. 159–198, 2007.

[58] Huang J., Voeten J., Florescu O., van der Putten P., and Corporaal H. *Predictability in Real-Time System Development*, chapter 8, pp. 167–183. Kluwer Academic Publishers, 2005.

[59] Huang J., Voeten J., Groothuis M., Broenink J., and Corporaal H. A model-driven design approach for mechatronic systems. In IEEE Computer Society, editor, *Proceedings of the 7th International Conference on Application of Concurrency to System Design – ACSD-07*, pp. 127–136. 2007.

[60] IEEE-1471. IEEE Recommended Practice for Architectural Description of Software Intensive Systems. IEEE Std 1471–2000, 2000.

[61] Jouault F. and Kurtev I. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*. 2005.

[62] Kapur D. and Mandayam S. Expressiveness of the operation set of a data abstraction. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 139–153. ACM Press, New York, NY, USA, 1980. doi:http://doi.acm.org/10.1145/567446.567460.

[63] Kent S. Model Driven Engineering. In *Integrated Formal Methods: Third International Conference, LNCS 2335)*, pp. 286–298. Springer Berlin / Heidelberg, 2002.

[64] Kernighan B.W. and Ritchie D.M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[65] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming, June 18-22*, pp. 327–353. June 18-22 2001.

[66] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.M., and Irwin J. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pp. 220–242. Springer, 1997.

[67] Kienhuis B., Deprettere E., Vissers K., and van der Wolf P. Quantitative Analysis of Application-Specific Dataflow Architectures. In *1997 International Conference on Application-Specific Systems, Architectures, and Processors (ASAP '97)*, pp. 338–349. IEEE Computer Society, 1997.

[68] Kleppe A.G., Warmer J., and Bast W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[69] Lang J. and Stewart D.B. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, volume 20(2):pp. 274 – 301, 1998.

[70] Lange C.F.J., Chaudron M.R.V., and Muskens J. In Practice: UML Software Architecture and Design Description. *IEEE Software*, volume 23(2):pp. 40–46, March 2006.

[71] Leavens G.T. and Clifton C. Foundations of Aspect-Oriented Languages Workshop. In *Foundations of Aspect-Oriented Languages Workshop*, volume 3. AOSD, 2004.

[72] Leavens G.T. and Clifton C. Foundations of Aspect-Oriented Languages Workshop. In *Foundations of Aspect-Oriented Languages Workshop*, volume 4. AOSD, 2005.

[73] Lewis G. and Wrage L. Approaches to Constructive Interoperability. Technical Report CMU/SEI-2004-TR-020 ESC-TR-2004-020, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.

[74] Lindig C. and Snelting G. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pp. 349–359. ACM Press, 1997.

[75] Lions J.L. ARIANE 5 Flight 501 Failure. Technical report, ESA/CNES, 1996.

[76] Lippert M. and Lopes C.V. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the International Conference on Software Engineering*, pp. 418 – 427. IEEE Computer Society, 2000.

[77] Littlewood B. Dependability assessment of software-based systems: state of the art. In *Proceedings of the International Conference on Software Engineering*, pp. 6–7. ACM Press, 2005. doi:http://doi.acm.org/10.1145/1062455.1062461.

[78] Lynch N.A., Merritt M., Weihl W.E., and Fekete A. *Atomic Transactions: In Concurrent and Distributed Systems*. Morgan Kaufmann, 1993.

[79] Nagy I. *On the Design of Aspect-Oriented Composition Models for Software Evolution.* Ph.D. thesis, University of Twente, June 2006.

[80] Nagy I., Bergmans L., and Aksit M. Composing aspects at shared join points. In R. Hirschfeld, R. Kowalczyk, A. Polze, and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*. Springer-Verlag, Erfurt, Germany, Sep 2005.

[81] van den Nieuwelaar N. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Ph.D. thesis, Technische Universiteit Eindhoven, 2004.

[82] Noonan L. and Flanagan C. Utilising evolutionary approaches and object-oriented techniques for design space exploration. In *Euromicro Conference on Digital System Design*, pp. 346–352. IEEE Computer Society Press, 2006.

[83] OMG. OMG. http://www.omg.org/cgi-bin/doc?ad/2002-4-10.

[84] OMG. OMG Unified Modeling Language Specification, Version 1.4. http://www.omg.org/docs/formal/01-09-67.pdf, June 2007.

[85] Potts C. Software-Engineering Research Revisited. *IEEE Software*, volume 10(5):pp. 19–28, September/October 1993.

[86] Ptolemy. Http://ptolemy.eecs.berkeley.edu/.

[87] Punter T. and Gool L. Experience Report On Maintainability Prediction at Design Level. Ideals technical report, Embedded Systems Institute, October 2005.

[88] Punter T., Voeten J., and Huang J. Quality of Model Driven Engineering. In *Model-Driven Software Development: Integrating Quality Assurance.* To appear.

[89] van der Putten P. and Voeten J. *Specification of Reactive Hardware/Software Systems - The Method Software/Hardware Engineering*. Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1997.

[90] Ramadge P. and Wonham W. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, volume 25(1):pp. 206–230, 1987.

[91] Robillard M. and Murphy G.C. Regaining Control of Exception Handling. Technical Report TR-99-14, Department of Computer Science, University of British Columbia, 1999.

[92] Roo A. *Towards More Robust Advice: Message Flow Analysis for Composition Filters and its Application*. Master's thesis, University of Twente, March 2007.

[93] Sabuncuoglu I. and Bayiz M. Analysis of reactive scheduling problems in a job-shop environment. *European Journal of operational research*, volume 126:pp. 567–586, 2000.

[94] Tau generation 2. Http://www.taug2.com/.

[95] The Open Group. IEEE Std 1003.1, The make utility. The Open Group Base Specifications Issue 6, 2004.

[96] Theelen B., Florescu O., Geilen M., Huang J., van der Putten P., and Voeten J. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 139–148. IEEE Computer Society, 2007.

[97] Toy W.N. Fault-tolerant design of local ESS processors. In *Proceedings of IEEE*, pp. 1126–1145. IEEE Computer Society, 1982.

[98] Tretmans G.E. *Tangram: Model-based integration and testing of complex high-tech systems*. Embedded Systems Institute, 2007.

[99] Viennot G.X. Heaps of Pieces, I: Basic definitions and combinatorial lemmas. In *Proceedings of the Colloque de combinatoire énumérative (UQAM 1985), Montreal, Canada*, volume 1234 of *Lecture Notes in Mathematics*, pp. 321–350. Springer, 1986.

[100] Wegner P. and Doyle J. Editorial: strategic directions in computing research. *ACM Computing Surveys*, volume 28(4):pp. 565–574, 1996.

[101] Weiser M. Program Slicing. *IEEE Transactions on Software Engineering*, volume 10(4):pp. 352–357, Jul 1984.

[102] van Wijk F., Voeten J., and ten Berg A. *An Abstract Modeling Approach Towards System-Level Design-Space Exploration*, chapter 22, pp. 167–183. Kluwer Academic Publishers, 2003.

[103] Zachmann. The Zachmann Institute for Framework Advancement. http://www.zifa.com.

[104] Zimmermann H. OSI Reference Model - The ISO Model of Architecture for Open System Communication. In IEEE, editor, *IEEE Transactions on Communication*, volume Com-28, Nr. 4, pp. 425–432. April 1980.