Towards an Expressive and Scalable Framework for expressing Join Point Models

Pascal Durr, Lodewijk Bergmans, Gurcan Gulesir, Istvan Nagy
University of Twente
{durr,bergmans,ggulesir,nagyist}@ewi.utwente.nl

ABSTRACT

Join point models are one of the key features in aspect-oriented programming languages and tools. They provide software engineers means to pinpoint the exact locations in programs (join points) to weave in advices. Our experience in modularizing concerns in a large embedded system showed that existing join point models and their underlying program representations are not expressive enough. This prevents the selection of some join points of our interest. In this paper, we motivate the need for more fine-grained join point models within more expressive source code representations. We propose a new program representation called a *program graph*, over which more fine-grained join point models can be defined. In addition, we present a simple language to manipulate program graphs to perform source code transformations. This language thus can be used for specifying complex weaving algorithms over program graphs.

1. INTRODUCTION

Current Aspect-Orientated Programming(AOP) approaches provide mechanisms to weave a set of (crosscutting) concerns with the base code. This is done at the design level like UML or lower levels like source code, byte code and, even the CPU instruction set. The exact locations that should be advised are specified through the use of a pointcut expression. This pointcut expression is subsequently mapped on a join point model. This model is usually an abstract representation of the base system code, for example a call graph or a control flow graph. These join point models are used to identify certain points in the static structure or in the execution of a program. This means that there can be a mismatch between the granularity of the join point models and of the code.

Most of the AOP approaches use a join point model similar to that of AspectJ[10]. The AspectJ join point model uses a dynamic call graph[15] to determine the correct join points. They thus provide commonly used primitive pointcuts like call, execution and field set and get. Such a model is sufficient for most cases. There are however situations where a more fine-grained join point model is required. Moreover, these models are mostly rooted in control flow and call graph information, but sometimes a pointcut expression based on data flow or data dependence is more suitable.

This paper proposes an expressive and scalable fine-grained framework to define join point models that incorporate both data and control flow analysis. One can subsequently specify a pointcut language which implements (part of) this join point model. Allowing different implementation strategies and language independence is the strength of this model.

The paper is structured as follows: First, we will motivate this research with two example concerns we encountered in our IDEALS[6] project with ASML[2]. This project aims at identifying, resolving and refactoring crosscutting concerns in embedded systems. Second, we will present our *program point model* and the source code representation we used to accommodate the model. Section 4 will propose a weaving language that operates on the presented program point model and uses the presented features. Finally, we will provide an overview of related work, conclusions and discussions.

2. MOTIVATION

Within our project we are currently developing a source to source weaver for the C programming language. We noticed that none of the current AOP approaches for C meet our requirements. Currently there are only two possible candidates for source code weaving of C code, AspectC[9] and AspectC++[20]. The first is a prototype that offer an AspectJ like mechanism for C. This does however not meet the level of granularity we require. The latter one supports weaving of C code but produces a mixture of C and C++ code after weaving. Arachne[11] is dynamic weaver for the C programming language. However the approach is limited to a specific platform and does not allow for static analysis by tools like Lint[16].

The subject system in this paper is an embedded system developed at ASML, the world market leader in lithography systems. The entire system consists of more than 12 million lines of C code and it has a number of typical crosscutting concerns. Within IDEALS[6] project we are working on two concerns that we present in the upcoming sections. However, we aim at a scalable framework model for other possible concerns as well. We therefore provide a comprehensive program representation and a weaving language that can enable developers to write more fine-grained and expressive pointcuts.

2.1 Parameter checking

Design by Contract[18] is a widely used technique to ensure the correct interaction between modules and components. A contract specifies a set of pre- and postconditions, that must hold before and after execution of a function or method. It thus specifies a set of invariants: conditions that must hold all the time. The checks to determine whether an invariant holds are usually written manually and are thus duplicated all over the code. The invariants are crosscutting since the same code is introduced in multiple places. In the ASML codebase, many of such checks are present to determine whether the value of a pointer parameter is *null*. ASML's coding conventions dictate that if a pointer parameter has the value *null*, this should be treated as an error and this error should be logged and returned to the caller. In addition ASML distinguishes between an input pointer parameters(referenced but not modified), and output pointer parameters(referenced and modified). Depending on whether a pointer parameter is input or output, a different error message is logged.

The code that checks the *null* pointer parameters is around 7% percent of the total code of certain components. This code is manually inserted and thus error prone. Our goal is to separate the parameter checking concern from the base system. However determining whether a pointer parameter is input or output requires data analysis of the code. We use AspectC++[20] as an example AOP approach on C(++)to show a possible solution using current AOP approaches. First, we show an example of a C function in the ASML's source code.

```
int CCXA_split_string(char *string, int string_length
    , char *head, char *tail)
{
   //...
}
```

Listing 1: Example ASML C function

This function takes a *string* and splits it into substrings, returning the *head* and the *tail*. The first two parameters are thus input (pointer) parameters and the latter two parameters are output pointer parameters. In order to adhere to the coding conventions we need to log different messages for the input and output pointer parameters. We define two advices, one advice checks the input pointer parameters and the other advice checks the output pointer parameters. The complete aspect definition is shown in listing 2.

```
aspect ParameterCheckingAspect
{
    pointcut CCXA_split_string_params (char *string,
         int str_length, char *head, char tail*) =
         execution("int CCXA_split_string(...)") &&
         args(string, str_length, head, tail);

    advice CCXA_split_string_params(string,
         str_length, head, tail) : around(char *string
         , int str_length, char *head, char *tail):
    {
        if(string == NULL)
    {
            ERROR_LOG("An input parameter error
                 occurred in function
                 CCXA_split_string, error code: %i\n",
                 INPUT_PARAMETER_ERROR);
            return(INPUT_PARAMETER_ERROR);
        }
        tjp->proceed();
    }

    advice CCXA_split_string_params(string,
         str_length, head, tail) : around(char *string
         , int str_length, char *head, char *tail):
    {
        if(head == NULL || tail == NULL)
        {
            ERROR_LOG("An output parameter error
                 occurred in function
                 CCXA_split_string, error code: %i\n",
                 OUTPUT_PARAMETER_ERROR);
            return(OUTPUT_PARAMETER_ERROR);
        }
        tjp->proceed();
    }
};
```

Listing 2: ParameterCheckingAspect

The initial motivation for separating concerns in this case was to reduce code size. However, the code size is not reduced as the number of checks is equal to the original unrefactored code and the number of lines used to write the check equals the number of lines used by the checks in the original unrefactored code. Furthermore, the programmer has to explicitly maintain the aspect specification while working on the base system code. Each time a new function is added, the developer should also write down the advice and pointcut definition. This example shows the need to incorporate data (flow) analysis into an AOP approach, as the type of the parameter can not be determined by solely looking at the call and control flow of a program. Current AOP approaches do not offer a mechanism to query the data flow of a program.

In our approach we can query the usage of global, local and parameter variables. Using such a mechanism we can write *one* pointcut designator to capture all functions where the parameters match a certain behavior. An AspectJ like example of such a pointcut is given in listing 3.

```
pointcut inputvars (Object p) :
      execution( int *(..))
        && hasParameter(p)
         && !parameterUsage(p,"* m *");
```

Listing 3: Example pointcut expression

The pointcut states that we are interested in the execution of any function with return type *int*. We constraint this set by indicating that it should have a parameter *p* which is not *modified*, indicated by *!parameterUsage(p,"∗m∗")*. We use here a regular expression to denote the behavior of this parameter. In this way we could also specify an output parameter(a parameter which is modified): *parameterUsage(p,"∗m∗")*. We could have chosen to annotate all the parameters with an input or output specification, this is however not supported by any AOP approach for C(++) and still requires explicitly specifying which parameters are input and which are output. The use of annotations is detailed in [19]. Our proposed approach can cope with changes made to the source code. For every new or changed function the data analysis is redone at weave time and the input and output parameters are calculated again. If the developer changes anything inside the function to make in input parameter an output parameter this is automatically captured by the pointcut.

The majority of existing AOP pointcut are based on control flow information. Whereas a pointcut mechanism based on the state and behavior of the data flow may capture the problem better.

2.2 Statechart updating

In [14], we discuss a statechart updating concern present in the ASML system. A number of statecharts express the behavior of ASML's lithography systems. At certain points in the program execution, one or more transitions are triggered and cause the affected statecharts to transition to the next state. The trigger should be inserted at the precise location, in the source code, designated by a sequence of statements.

The statechart updating concern motivates the need for more fine-grained join point mechanisms. There are cases where high level pointcuts like call and execution are not sufficient to identify the correct join point.

3. PROGRAM REPRESENTATION

Our main objective is to define a program representation model that supports a fine-grained join point model, can represent properties about the program—including the results of code analysis such as data and control flow—and can be applied for multiple languages (i.e. the *model* must be language-independent). In this section we will propose a model that addresses these requirements.

3.1 Program points

First, we distinguish a *concrete join point*; a particular point in the program that is used for weaving in additional behavior, and a *potential join point*; a point in the program where additional behavior *can* be woven in. We will use the term *weave point* for designating potential join points, and *join point* for a particular point in the program that has been selected for inserting additional behavior. Unfortunately, in the literature, both meanings of the term join point seem to be used interchangeably. In particular, the term *join point model* (JPM), which describes the characteristics of the potential join points, is often used. We will adopt this commonly used term as well, although for consistency, *Weave Point Model* would be more appropriate.

Second, we would like to point out that the granularity of the join point model need not be the same as the granularity of the program representation. The reason is that a pointcut specification may need more detailed information about the program to decide which weave points are selected. For example, in AspectJ-like languages, the program representation distinguishes individual arguments of methods, whereas these are not available as join points. In these languages, primitive pointcuts can be used within pointcut designators to access such additional information about the program. For this reason, we introduce the notion of *program points*, which are defined according to a *program point model* (PPM). A program point is an atomic element in the representation of the program. A program is a sequence of statements in a programming language. What to consider 'atomic' is a matter of design decisions while defining the aspect language and corresponding program point model. Typical examples of program points are statements for: assignment, call, control, function (or method) start and end and variable declaration, etc. Table 1 shows a set of possible program points for the C programming language. This list is taken from [4]. Such a list can be defined for any language.

Program point:	**Description:**
Call-site	Direct call to a function
Indirect-call	Indirect call via a function pointer.
Expression	Expression or assignment statement.
Control-point	If, switch, while, for, etc...
Declaration	Declaration of a local or global variable.
Variable-init.	Initialization of a variable.
Return	Return statement.
Actual-in	Actual parameter of function call.
Actual-out	Return value of a function call.
Formal-in	Formal parameter of a function
	definition.
Formal-out	Return value of a function definition.
Entry	Execution start of a function.
Exit	Point just before returning to the caller.
Jump	Goto, break, or continue.
Switch-case	Case or default.

Table 1: Possible program points for C

As we explained before, in some join point models, not all available program points can be used as a join point. Again, which program points to include as weave points is a design decision in developing an aspect language and its corresponding join point model. The relation between program points, weave points and join points is expressed by:

$(concrete)\ join\ points \subseteq weave\ points \subseteq program\ points$

in other words, all concrete join points are also weave points (i.e. potential join points), and all weave points must be program points (but not necessarily the other way around).

3.2 Program graph

Our program representation model uses program points as its core, and constructs a full program representation by (a) adding properties to program points, and (b) adding information about the relations between the program points. To this end we adopt a graph representation, where the nodes correspond to program points, and the edges express relations between program points. We assume the nodes are labeled and can have properties[1].

To represent the properties of a program, we distinguish three kinds of relations between program points; control flow, call and structural.

- The *control flow* relations show in which sequence the program points can be executed. This information can be gathered through the use of standard control flow analysis. This analysis is assumed to be done for each function and thus resulting in a set of lattices of program points and directed control flow edges.
- The *call* edges link the separate control flow lattices via directed edges between the call program point and the return of this call. This information can also be derived via control flow analysis.

[1]Note that these properties can also be modeled as separate nodes with directed—appropriately labeled—edges pointing from the main node

- The *structural* edge is used for representing the static program structure and other dependencies. Examples are a dependency to a superclass (i.e. inheritance relation), a relation between a package and a class that it contains, and the definition of a parameter of a function. This structural information can be extracted from the AST.

In our join point model we also want to be able to use annotations. For the motivation for using annotations and more information regarded the annotations we refer to Nagy[19]. Annotations can be attached to almost every possible statement and thus to almost every possible program point. We model these annotations as properties of the program point. We state that each program points has a set of annotations. The annotations can be extracted from the AST.

As motivated by the parameter checking concern we want to be able to express data oriented pointcuts. This requires the presence of this information inside the PPM. We state that each program point has a set of referenced global and local variables and parameters and a set of modified global and local variables and parameters. These sets can be derived through the use of data flow analysis.

The above mentioned program points, edges and attributes of the program points form one large program graph. We now give a formal definition of the presented program graph. The program graph of program P is defined as follows: $G_P =< N, E_{flow}, E_{call}, E_{structure} >$. Where:

- N is the set of program points. N has the following attributes:
  - A is the set of annotations attached to this node.
  - R is the set of global and local variables and parameters being referred(read) to in this node.
  - M is the set of global and local variables and parameters being modified(written) in this node.
- E_{flow} is the set of edges representing the control flow from one program point to another. There is an edge from p_1 to p_2 if the program point p_2 can be executed immediately after program point p_1.
- E_{call} is the set of edges representing the call to a function and the return of all calls.
- $E_{structure}$ is the set of edges representing the context relation.

To illustrate the concepts introduced above we first show an example C function of ASML, *CCXA_split_string*, and then the equivalent program graph, $G_{CCXA\_split\_string}$. The function is simplified and names have been renamed due to confidentiality agreements and space considerations. This function will serve as a running example to illustrate various aspects of the PPM.

```
int CCXA_split_string(char *string, int string_length
    , char *head, char *tail)
{
    int result = OK;
    result = head_string(&head, &string, string_length
        );
    if(result != OK)
    {
        ERROR_LOG("An error occurred in function
            CCXA_split_string, error code: %i\n",
            result);
    }
    else
    {
        result = tail_string(&tail, &string,
            string_length);
        if(result != OK)
        {
            ERROR_LOG("An error occurred in
                function CCXA_split_string, error
                code: %i\n",result);
        }
    }
    return result;
}
```

Listing 4: Example C function

Figure 1 shows a simplified version of the program graph of the example in 4. The edges are labeled with the type and we omitted the conditional labels like *true* and *false*. The two top program points(PP) and the edges between

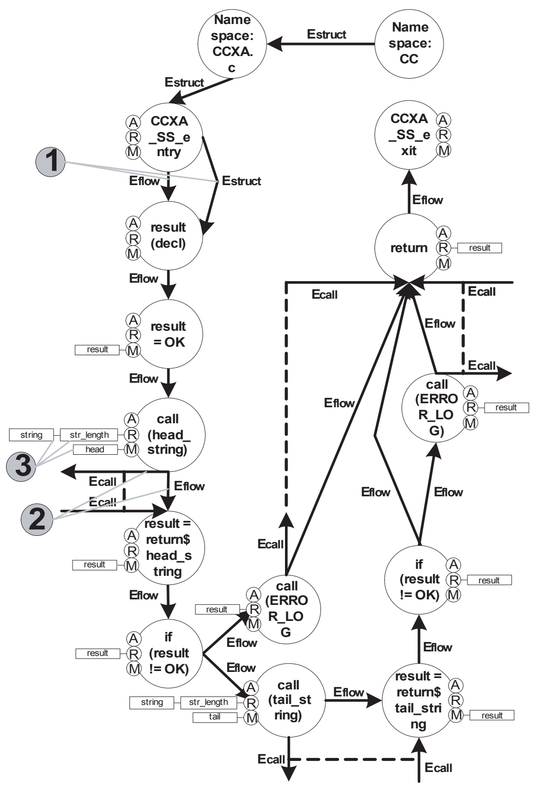

Figure 1: Program graph

them show the namespace of function *CCXA_split_string*, this could be a representation of a directory and file structure. See (1) in the graph. The start of the function is represented by the PP *CCXA_SS_entry*. This PP has two outgoing edges. One control flow edge to the *result(decl)* PP stating that the result is declared immediately after entering the function. The structure edge indicates that the function *CCXA_split_string* declares a local variable *result*.

The *call(head_string)* PP has two outgoing edges, shown near (2) in the graph. The end of the call edge is attached to the entry point of the function it calls. The return of this function is represented by the incoming call edge of the *(result = return\$head_string)* PP. The start of this edge is attached to the exit point of the called function. We model the control flow from the call PP to the return PP explicitly. This can be seen near (2) in the graph.

Each program point has properties indicating the set of annotations attached to this PP, a set of referenced variables or parameters and a set of modified parameters and variables. The referenced set of a function entry PP is the union of all reference sets of all PPs in the control flow paths starting at the entry PP and ending at the exit PP. Similar, the modified set of a function entry PP is the union of all modify sets of all PPs in the control flow paths starting at the entry PP and ending at the exit PP. This means that the referenced and modified set of a call PP will also contain the sets of the function that is called[2].

3.3 Recovery example

To illustrate the program graph we provide an example for a simple recovery concern. The recovery concern should assign the *OK* value to the result variable in the case the *head_string* function returns an error code. We thus only want to insert the recovery advice after the *head_string* call but not after the *tail_string* call.

A pointcut designator specifying the call to *ERROR_LOG* would result in accidental join points. We thus have to indicate the exact join point through the use of a a sequence of program points. For this example the sequence would contain a program point path starting with a *call(head_string(..)* PP immediately followed by an *if* PP, and within then branch program point a *call(ERROR_LOG)* PP. Such a pointcut designator could be defined as a regular expression.

We subsequently have to match this sequence on the PPM. This mapping can not be made directly on the program graph. We accomplish this by transforming the lattices of program control flow graphs representing the functions to a labeled transition system. This is achieved by adding the labels of the PP to the incoming edges, we can now transform the program graph to an automaton. The start and end state of the automaton are the entry and exit point of the function. The resulting automaton can be used to match the regular expressions representing the pointcut expressions. It should be noted that the above example pointcut designator is very expressive, and one could define a higher level pointcut language which abstracts from this model.

3.4 Parameter checking example

In section 2.1 we showed a example pointcut in listing 3. The pointcut expression states that for each parameter of a function with return type *int* which is not modified we want to insert the input point parameter checking advice. We can translate this also to a regular expression on the program point automaton. However we can define constraints that must hold on all possible join points given the regular expression. In this case all paths from the entry to the exit program points of each function that returns an *int*. In the example we specified two predicates. The first predicate binds each parameter to *p*. The second predicate constraints this set even further by stating that if such a parameter exists then it may not be modified. This can be translated to program graph by iterating over all modified sets of all program points of all functions with return type *int* and determining whether the parameter *p* is an element in such any modified set. If this is that case the parameter is written and it is not in input pointer parameter but an output pointer parameter. All parameters that are not in the any of modified sets are thus input pointer parameters.

4. WEAVING LANGUAGE

We primarily focused on the program point model and the motivation for this model. In this section we propose a weaving language to complement the program point model.

Our definition of a join point is a path $path_{jp}$ in the program graph. This join point is defined as a tuple of program points $< PP_{jps}, PP_{jpe} >$ where PP_{jps} denotes the start, and PP_{jpe} the end of the join point path. We can thus identify a single program point by stating that PP_{jps} is equal to PP_{jpe}. We define an advice as a path $path_{adv}$ denoted as a tuple $< PP_{advs}, PP_{adve} >$, similar to the join point tuple. Given the join point and advice tuples we define operations on the program graph G_p containing $path_{jp}$.

InsertBefore : Inserts $path_{adv}$ before the join point $path_{jp}$. This is achieved by detaching all incoming control flow and call edges of PP_{jps} and attaching these edges to PP_{advs}. We then add a control flow edge from the PP_{adve} to the PP_{jps}.

InsertAfter : Inserts $path_{adv}$ after the join point $path_{jp}$. This is achieved by detaching all outgoing control flow edges of PP_{jpe} and attaching these edges to PP_{adve}. We then add a control flow edge from the PP_{jpe} to the PP_{advs}.

Substitute : Substitutes $path_{jp}$ with the $path_{adv}$. This achieved by detaching all incoming edges from PP_{jps} and attaching them to PP_{advs}. Similarly we detach all outgoing edges from PP_{jpe} and attach them to PP_{adve}.

Remove : Removes $path_{jp}$ from G_p and restores the control flow by attaching all incoming control flow and call edges of N_{jps} to all out outgoing edges of PP_{jpe}.

We note that the advice does not have to include the start or end node. The operations can specify whether or not to include the start or end node of the join point and advice paths. This allows us to use any possible graph structure as an advice, no matter what structure; a single edges, a single node or more complex graph structures.

We must ensure that after weaving all the advices we are still left with a valid program. We thus have to constrain the operations and the possible join point and advice tuples. The advice can contain language constructs which are not allowed at the location of weaving. For example inserting a declaration of a local variable just before the *return* statement on line 17 in listing 4. Such insertion will result in an erroneous program.

The operations *Substitute*, *Remove* and *Cut* can only be applied if the join point path is confined to the control flow graph of a single function. If the PP_{jps} and the PP_jpe are not located in the same function we have no way of determining whether the *Substitute*, *Remove* and *Cut* will brake the two functions it spawns.

[2]It should be noted that in Object Oriented language this is not that simple due to polymorphism.

After all operations we need to make sure that all program points are still reachable, via control flow and call edges. If there are program points with no incoming control flow and call edges we encountered a set of PPs which can not be reached and are therefore dead code. It could also be the case that structural edges are left dangling, for example if we removed a variable declaration or even an entire function. All these structure dangling edges should be removed. This part of our research is still in the early stage and will require future work.

We discussed some of the constraints and possible problems briefly. We will not focus on these constraints and problems in this paper but we are aware of these possible issue.

5. RELATED WORK

Many of the current AOP approaches currently available are based on the established join point model of AspectJ[10]. Examples of these approaches, besides AspectJ, are AspectC[9, 8], AspectC++[20], AspectWerkz[3] and JBoss[7]. They lack the fine-grained expressiveness that our proposed model offers. Furthermore, the join point models used by these approaches are coarse-grained, i.e. call, execution, global get and set. Our approach is more fine-grained and allows us to specify pointcut expressions that capture some concerns[14] better.

None of mentioned approaches incorporate data analysis into their join point model. This may result in hard to write and maintain pointcut expressions as demonstrated in section 2.1. [17] introduces the *dflow* primitive pointcut for AspectJ. This pointcut provides a way to determine a join point based on the data flow of the parameters bound by the *args* or *return* pointcut. This approach is however limited to the parameters of a function and does not reason about the data flow of local and global variables. Their motivation to use of data oriented pointcuts is illustrated by a *sanitizing* task within a web server environment. Therefore none of the current AspectJ-like approaches provide the degree of data oriented pointcuts we offer.

Event Based AOP (EAOP[1]) is an AOP approach that offers similar features. Event base AOP inserts event triggers in the base system code. Advice can be attached to these events which is executed once such an event is fired. Similar to our approach this approach offers the possibility of providing a sequence of primitive pointcut. These sequences are however limited to relative simple orderings, our approach can express more as it uses regular expressions over the program points. This provides us with the a more flexible and expressive way to define pointcuts. They have an implementation of EAOP on C called Arachne[11]. In this approach they weave the advice in the binary code of a compiled C program. At weaving time every assembly instruction that performs a event will be replaced by a call to a hook. The hook will ensure that the advice is executed. This approach rewrites IA-32(x86) instructions and is thus platform dependent.

GrammaTech[5] and the University of Wisconsin did a preliminary study on the possibility of using the CodeSurfer[4] tool to provide a framework for AOP of embedded systems[13]. This work studied the use of dependence graphs in an AOP environment. They also envisioned pointcuts using data flow analysis. The proposed join point model and pointcut designators is AspectJ-like and thus does not offer the level of granularity and expressive offered by our approach.

6. CONCLUSIONS AND DISCUSSION

We have motivated the need for a fine-grained join point model and we showed how one could use such a model to achieve more expressiveness in these models. We illustrated in section 2.1 based on a real example that the results of data analysis may be needed to express certain pointcuts as well. And we argued that our goal is to define a scalable, open-ended model for representing programs, such that we can reason about and manipulate programs at a fine-grained level.

In short the presented framework has the following characteristics

- A very fine grained join point model.
- The program graph integrates information from the Abstract Syntax Tree with call, control flow and data dependency graphs.
- The model is scalable in that it can express both simple, coarse-grained join point models and fine-grained join point models with a rich program representation.
- We defined a small set of operations over our program graph that are both primitive, expressive and maintain important consistency constraints of the program graph.

The work we presented in this paper still has many (open) issues, we discuss a few of these:

Fragile join points : It should be noted that using the static structure of the program will lead to fragile join points. These fragile join points are the result of the high coupling between aspect and base system. If the source code changes these kinds of join point will likely be invalidated. However because we have the expressiveness to leave out superfluous details in the pointcut descriptor that are not relevant for the join point we can reduce the fragility of the join points.

Program graph creation : We did not cover the creation of program point graph from source code. As our program graph integrates information from the Abstract Syntax Tree with call, control flow and data dependency graphs we should have the ability to build up these representations from the source code. There are tools like CodeSurfer[4] that provide advanced control and data flow (dependence) analysis.

Program Normalization It should be noted that we assume the program to be normalized[21]. The normalization process makes identifying the joint point easier as non composed expressions are present in the code. On the other hand, it is important that the model that the programmer has about the program is not different (i.e. does not lead to different join points) from the normalized model.

7. ACKNOWLEDGMENTS

We would like to thank Remco van Engelen for his valuable input and for proofreading drafts of this paper, and all members of the Ideals project team, for input about the topic.

This work has been carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program.

8. REFERENCES

[1] Event based aop. http://www.emn.fr/x-info/eaop/.

[2] ASML. http://www.asml.com.

[3] AspectWerkz. http://aspectworkz.codehaus.org/.

[4] CodeSurfer. http://www.grammatech.com/products/codesurfer/.

[5] GrammaTech. http://www.grammatech.com.

[6] Idiom Design for Embedded Applications on Large Scale. http://www.embeddedsystems.nl/ideals/.

[7] JBoss. http://www.jboss.org/.

[8] Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003.

[9] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98. ACM Press, 2001.

[10] A. Colyer. AspectJ. In Filman et al. [12], pages 123–143.

[11] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Sgura-Devillechaise, and M. Sdholt. An expressive aspect language for system applications with arachne. In *Proceedings of the 4th international conference on Aspect-oriented software development*, Chicago, USA, Mar. 2005. ACM Press.

[12] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development.* Addison-Wesley, Boston, 2005.

[13] GrammaTech. A framework for aspect-oriented programming of embedded systems. http://www.grammatech.com/research/darpa-sttr-2000/index.html.

[14] G. Gulesir, L. Bergmans, and P. Durr. Separating and managing dependent concerns. In *submitted to LATE '05: The 1ste Workshop on Linking Aspect Technology and Evolution*, Chicago, USA, March, 14 2005.

[15] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[16] S. C. Johnson. *Lint, a Program Checker, in Unix Programmers Manual.* AT&T Bell Laboratories: Murray Hill, 1979.

[17] K. Kawauchi and H. Masuhara. Dataflow pointcut for integrity concerns. In *AOSDSEC '04: The Workshop on AOSD Technology for Application-level Security*, Lancaster, UK, March, 23 2004.

[18] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., 1988.

[19] I. Nagy and L. Bergmans. Towards semantic composition in aspect-oriented programming. In *1st European Interactive Workshop on Aspects in Software*, Sep 2004.

[20] O. Spinczyk, A. Gal, and W. Schroder-Preikschat. Aspectc++: An aspect-oriented extension to the c++ programming language. In J. Noble and J. Potter, editors, *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of *Conferences in Research and Practice in Information Technology*, pages 53–60, Sydney, Australia, 2002. ACS.

[21] E. Visser. Program normalization. http://www.program-transformation.org/Transform/ProgramNormalization.