Using Aspect-Orientation in Legacy Environments for Reverse Engineering using Dynamic Analysis

—An Industrial Experience Report—

Bram Adams [a] Kris De Schutter [b] Andy Zaidman [c]
Serge Demeyer [d] Herman Tromp [a] Wolfgang De Meuter [b]

[a] *Ghent University*
{*Bram.Adams, Herman.Tromp*}*@ugent.be*

[b] *Vrije Universiteit Brussel*
{*kdeschut, wdmeuter*}*@vub.ac.be*

[c] *Delft University of Technology*
a.e.zaidman@tudelft.nl

[d] *University of Antwerp*
Serge.Demeyer@ua.ac.be

Abstract

Legacy systems evolve, and as they do they need to be maintained and re-engineered in order to remain of value to their owners. We report on reverse engineering such a legacy industrial system (407 C modules holding 453KLOC, supported by 269 makefiles). We apply dynamic analysis in order to regain insight into the system. Aspect oriented programming (AOP) is used for instrumenting the system and for gathering the data. We show a comparison of available AOP tools and argument our choice. This approach works and is conceptually very clean, but comes with a major quid pro quo: integration of AOP tools with the build system proves an important issue. This leads to the question of how to reconcile the notion of modular reasoning within traditional build systems with a programming paradigm which breaks this notion. The results of the dynamic analysis itself prove accurate and useful, and have been validated by the original developers. [1]

Key words: dynamic analysis, aspect-oriented programming, industrial case study, program comprehension, C

[1] This article is an extension to our earlier paper *Regaining Lost Knowledge through Dynamic Analysis and Aspect Orientation*, published in the proceedings of the 10th European Conference on Software Maintenance And Reengineering (CSMR'06) [79].

1 Introduction

Legacy software is omni-present: software that is still very much useful to an organization – quite often even *indispensable* – but the evolution of which becomes too great a burden [3]. This burden can be caused by an increase in complexity brought on by the normal evolution of the system [69,7,53,20,21,48]. Classic symptoms include:

- a lack of experienced developers or maintainers,
- a lack of up-to-date documentation, and
- technology that does not reflect the current (business) environment.

To counter this phenomenon, a number of solutions to cope with evolution have been proposed [3,67] in the field of re-engineering [14]. When applying these countermeasures in a reliable, economically sound and swift fashion, the software engineer would ideally like to have (1) a *deep insight* into the application in order to start his/her reengineering operation [68,20,47] and (2) a well-covering (set of) regression test(s) to check whether the adaptations made are behavior-preserving [21,23]. In practice, legacy applications seldom have up to date documentation [53], nor do they have such a set of tests.

For all these reasons we are interested in the re-engineering of legacy E-type systems ("*software systems that solve a problem or implement a computer application in the real world.*" [46]). Recent research [52,17,45] suggest that aspect oriented programming (AOP) [40] plays an important role in this effort as it provides a modularised way to change the existing behaviour of a system without having to destructively modify that system's source code in any way. The modularity provides us with opportunities for re-engineering, while the non-invasiveness takes care of some of the psychological concerns associated with modifying business-critical source code.

We have been looking at applying AOP in forward engineering [44,66,65,2], as have others [8,10,52], with success. Different from these, this paper takes a first look at an opportunity for AOP in a reverse engineering setting. This is an essential first step in the re-engineering process and has been reported to take up to 60% of the required effort [18].

Considering the industrial setting that we are working in, we choose to use dynamic analysis for our reverse engineering process. This choice is instigated by the fact that dynamic analysis allows us to follow a goal-oriented strategy, i.e. it lets us analyze only those parts of the system that we are really interested in [78]. This goal-oriented strategy is certainly warranted considering the scale of the legacy application. Furthermore, it puts us in the unique position to report on the benefits of using dynamic analysis in a large-scale industrial legacy setting, of which no other reports are known to us. *[Reviewer 3 says:*

Eisenbarth?]

In order to enable this dynamic analysis, we introduce a simple tracing aspect into the system. Given that we only need to collect a representative trace of the running application in order for the dynamic analysis to work we could also have opted for dedicated tools such as DTRACE [13] or ATOM [72]. There are two reasons that we do not do this. One is that we are looking at AOP as a tool in the *entire* re-engineering chain and *not* limited to a particular reverse engineering technique. As proposed by De Roover et al. [64], aspects can generate *reverse*-engineering results in such a way that *re*-engineering aspects can exploit these results to steer their re-engineering tasks. In this respect AOP is more interesting as it is more generally applicable than the aforementioned tools. The second reason is that we also need to consider how to get our aspects applied in real-life systems. As this paper shows, even for something as simple as a tracing aspect, *this is not trivial*. Indeed, as the prototypical example of an extremely scattered aspect, a tracing aspect actually provides us with something of a stress test with respect to the support of aspects in the legacy system.

As part of our research in the ARRIBA [2] project our focus is on legacy systems written in (some dialect of) the C programming language. The experiment reported on in this paper is therefore on a mid-size real-life system which has accumulated a mix of Kernighan & Ritchie [3] (K&R) as well as ANSI-C style code. This has an impact on our choice of AOP tool, which this paper will also take into careful consideration.

In short, the contributions of this paper are:

- the application of a dynamic analysis on an industrial legacy application,
- a comprehensive overview of AOP tools for the C programming language,
- the introduction of a new AOP tool which fits our re-engineering goals,
- a discussion of some of the problems found when applying AOP in a legacy setting.

The structure of the remainder of this paper is as follows: section 2 first elaborates on our choice of reverse engineering technique to be applied to the system. Section 3 then follows up by exploring which AOP tool to apply. As we will see there is none that fits the bill and so section 4 introduces a tool of our own which has been created according to our re-engineering goals. Next, section 5 shows the actual experiment, including the aspect we apply, the results we get from the analysis, and the validation of those results with the system's developers. Section 6 then discusses the problems encountered

[2] *"Architectural Resources for the Restructuring and Integration of Business Applications"* More info on this project at `http://arriba.vub.ac.be/`.

[3] After their seminal work [38].

while trying to apply AOP to this system. Related work is shown in section 7, followed by section 8 which rounds up the discussion with our conclusions.

2 The Case

[Reviewer 1 says: part of section 5 to here] While the concept of dynamic analysis goes back to at least the early seventies [4], there has recently been a renewed interest in dynamic analysis techniques that deal with large-scale program comprehension [31,36,81,80,60,73]. This renewed interest can partly be explained by the increasing need to understand large scale object-oriented software. This object-oriented software makes abundant use of polymorphism and the late binding mechanism makes it hard to understand the software when only using static analysis techniques.

One such dynamic analysis technique, the *key class identification* technique was developed in-house and was previously extensively validated with object-oriented software systems [80]. This previous study has shown that the key class identification technique is able to find those classes in a system that need to be studied during early program comprehension phases. We found it particularly worthwhile to verify whether this key class identification technique could also prove its worth in non-object-oriented legacy systems where it would identify key modules rather than key classes.

This section discusses some of the technicalities of the key class identification technique.

2.1 Dynamic coupling based analysis

Within software systems, the concept of coupling is inevitable as program parts —be they classes or modules— work together to reach a certain goal. Classes that exhibit a relatively high level of coupling can be designated "*influential*". Influential, because they have a certain amount of control over *what* the application is doing and *how* it is doing it. A similar observation of influential classes was made by Tahvildari in her research about design flaws [74]: "Usually, these most important concepts of a system are implemented by very few key classes, which can be characterized by a number of properties: (1) they manage a large amount of other classes or use them in order to implement their functionality, (2) they are tightly coupled with other parts of the system and (3) they tend to be rather complex, as they implement much of the legacy system's functionality." As such, structural dependencies between modules of a system can indicate modules that are interesting for initial program

comprehension [62].

To be more specific, the key class identification technique uses run-time export coupling, which is a measure for the degree to which a module requests other modules to do work for them (delegation). This gives us all actual dependencies that happen at run-time, provided we have a well-covering execution scenario. However, coupling measures are typically between two classes or modules, whereas we want to take into consideration the complete structural topology of the application. To overcome this strict binary relation between modules, we add a transitive measurement for reasoning over the topology. We use webmining techniques for this [80].

2.2 Webmining analysis

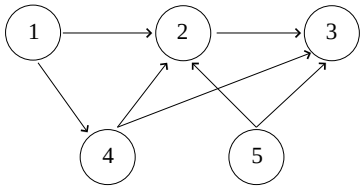
Webmining, a branch of datamining research, analyzes the topological structure of the web trying to identify important web pages based solely on their hyperlink structure. By replacing the hyperlink structure of the web (i.e., a web graph) by a call graph that shows the structural dependencies of a software system, we are able to use the same basic technique to retrieve important classes or modules.

The HITS webmining algorithm [42] identifies so-called *hubs* and *authorities* in (web) graphs. Conceptually, hubs are nodes that have a high number of outgoing edges, while authorities are nodes that have many incoming edges. In terms of the Internet, hubs are pages that mainly have a referring function, e.g., web directories, lists of personal pages, etc. On the other hand, an authority contains useful and/or highly detailed information regarding a specific subject. In terms of program comprehension, hubs are modules that contain the core high-level logic of the application (conceptually, these are the influential classes we talked about in Section 2.1), while authorities are implementers of more low-level functions (e.g., utility classes [36]).

The basic mechanism for the HITS algorithm is as follows: every node in the graph i gets assigned to it two numbers; a_i denotes the authority of the page, while h_i denotes the hubiness. Let $i \rightarrow j$ denote that there is a calling relationship between modules i and j, and let $w[i,j]$ be the number of different methods of module j called from within i, i.e., the *weight* (or importance) of the calling relationship. The recursive relation between authority and hubiness is captured by formulas (1) and (2).

$$h_i = \sum_{i \rightarrow j} w[i,j] \cdot a_j \tag{1}$$

$$a_j = \sum_{i \rightarrow j} w[i,j] \cdot h_i \tag{2}$$



(a)

Iterations	Nodes 1	2	3	4	5
1	(1,1)	(1,1)	(1,1)	(1,1)	(1,1)
2	(2,0)	(1,3)	(0,3)	(2,1)	(2,0)
3	(4,0)	(3,6)	(0,5)	(6,2)	(6,0)
4	(8,0)	(5,16)	(0,15)	(11,4)	(11,0)
5	...	...	...	...	...

(b)

Fig. 1. Example graph and the accompanying first iterations of the HITS webmining algorithm. *[Reviewer 1 says: add row with final values algorithm]*

The HITS algorithm starts with initializing all h's and a's to 1, and subsequently repeatedly updates the values for all pages using the formulas (1) and (2). If after each update the values are normalized, this process is known to converge to stable sets of authorities and hub values [42]. The h and a values are normalized so that: $\sum_{\forall h_i} (h_i)^2 = 1$ and $\sum_{\forall a_i} (a_i)^2 = 1$. *[Reviewer 1 says: no forall]*

As an example consider the graph given in Figure 1. The table in this figure shows three iteration steps of the hub and authority scores (represented by tuples *(H,A)*) for each of the five nodes from the graph in Figure 1(a). From this, we can conclude that 2 and 3 will be good authorities as can be seen from their high A scores in Table 1(b). Looking at the H values, 4 and 5 will be good hubs, while 1 will be a less good one [4].

The result set obtained from this heuristic is a list of all the modules of which containing procedures were executed during an execution scenario. These modules are ranked from being important to being irrelevant during early program comprehension phases. An earlier study on Java software showed that the technique is able to retrieve the most important classes within a system with a level of recall of 90% and precision of 60%. More information about this technique and its validation can be found in [80].

2.3 Requirements for applying the dynamic analysis

In order to make this dynamic analysis work, the following is needed:

- a well-covering execution scenario in order to obtain a representative result set.

[4] Be aware that the example from Figure 1 uses integer values for calculating the hub and authority scores, while the actual implementation uses floats due to the normalization of the actual scores.

- a sufficiently fine-grained trace. In the context of C this means individual procedure calls. We also need context information, in particular on the relevant source modules.
- information about the call- and return-sequences, in order to get an accurate picture of the dynamic behavior of the applications.

3 AOP tools for legacy C applications

As discussed by Mens et al. [52], aspect extraction and evolution are two crucial activities when re-engineering a system using aspects. Failure or success of AOP for re-engineering depends to a large extent on sufficient aspect language support. Without this, the re-engineered system risks becoming unmaintainable and even less manageable than the original system.

3.1 Aspect Oriented Programming

Aspect oriented programming (AOP) modularises so-called "crosscutting concerns" (CCCs) [40]. When developers implement these concerns using traditional programming language techniques, two undesired phenomena typically crop up in the source code: scattering and tangling. The former corresponds to implementation fragments of a concern (like e.g. caching) which occur at many places throughout the source code. Changes to the concern's implementation likely require to make changes at many places in the system, which is tedious, error-prone and hampers understandability. The situation is even worse, because at each location where a caching concern fragment occurs, it can be tangled (mixed) with fragments of other concerns. This means that programmers need to understand the interplay between multiple concerns before being able to modify the caching concern. AOP deals with these undesirable program properties by extracting crosscutting concerns in a new kind of modules: aspects.

To date, AspectJ is still the primary aspect language in existence, both in research and in practice. This is an aspect language for Java which has introduced the concepts of advice, pointcut, join points, etc. An aspect is similar to a class or module, but can contain "advice", which consists of a "pointcut" [5] and an "advice body". According to the most common school ("asymmetric AOP"), the implementation of crosscutting concerns is extracted from the "base code". The latter corresponds to the implementation of the main concerns, the so-called "dominant decomposition" which forms the backbone of

[5] Sometimes abbreviated to "PCD", for "pointcut designator".

the whole system. CCC implementation fragments are separated from the base code and localised into (possibly) multiple advice bodies of an aspect.

Code separation is only one part of the effort required to resolve scattering and tangling. One still needs to specify at which moments during the base program execution an advice body should be invoked. Instead of embedding explicit calls to advice within the base code, an advice is invoked automatically once a condition (pointcut) is satisfied. This inversion of dependencies [56] forms the core idea behind AOP. The moments in time when advice can be triggered are called "join points", as this is where the main concern(s) and a CCC join each other. Established kinds of join points are method calls and executions, variable access and manipulation, etc. A pointcut can make use of program structure, name patterns, dynamic program state, etc. to describe the intended set of join points. The process of matching join points with a pointcut and of executing advice on a pointcut match is called "weaving". Conceptually, a "weaver" monitors the program execution and checks each join point to decide whether there is a match or not. In practice, the set of interesting join points can be reduced based on analysis of the pointcuts, or weaving can be moved completely to the compiler, with only a couple of dynamic checks ("residues") left at run-time. The only restriction is that the developer should always have the perception of a run-time monitor.

Some aspect languages like AspectJ also provide means for managing static crosscutting concerns, i.e. inter-type declarations [6] (ITD) [39]. Whereas advice alters program behaviour, ITD alters types or may facilitate program verification and error handling. The latter two applications solicit compiler feedback if a user-specified pointcut matches during weaving. Type alteration allows classes and interfaces to be extended with new attributes or methods and may even change the inheritance hierarchy by adding new interfaces to be implemented or changing the superclass. The idea is that these structural modifications support behavioural CCCs implemented as advice, but that they also allow base code developers to explicitly use the introduced attributes or methods. Griswold et al. call this "language-level obliviousness" [33], i.e. developers are aware of the woven aspects. If developers do not know anything about the possible aspects, one speaks of "designer obliviousness", unless developers may prepare the base code to expose better join points ("feature obliviousness").

Commonly, a distinction is made between "homogeneous" and "heterogeneous" CCCs [17]. Homogeneous concerns are said to look almost identical everywhere they occur. On the other hand, heterogeneous concerns may vary widely between different occurrences. As a consequence, the implementation of homogeneous concerns may easily be localised into one advice body, whereas

[6] The original name for this feature was "introduction".

heterogeneous concerns are harder to implement in a robust way. In the latter case, the advantages of AOP may seem to be limited, but this actually depends on the expressivity of the aspect language, i.e. the advice and pointcut language. The better variability can be expressed in the aspect language, the easier heterogeneous advice can be extracted into advice.

AOP has especially been studied in the context of OO systems, as a means to overcome the problems of scattering and tangling in even the most advanced OO languages. Nevertheless, CCCs are more fundamental than this. Anytime a problem is tackled by making some structural design decisions, remaining concerns have to fit into this main decomposition somehow. This problem is named the "tyranny of the dominant decomposition" [75]. Hence, CCCs not only occur in OO systems, but also in procedural or functional programs [40,45,65], as these also start from a main decomposition of the system. Keeping in mind that OO languages offer more powerful composition constructs than modular or procedural programming, this means that the latter have even less means to manage CCCs. Research has shown [12,17,9–11,8] that CCCs represent an important evolution problem in legacy systems, especially if one takes the scale of these systems into account (millions of lines of code). Tangling and scattering of CCCs with the main concern heavily impact program understandability, while scattering increases the cost of maintenance and reduces traceability of code fragments to the modeled concern. Various researchers have considered AOP as a viable solution to deal with this problems in legacy systems [65,55,8]. This paper investigates this claim.

3.2 Requirements for aspect languages for legacy systems

Finding the right aspect language for re-engineering legacy systems is not an easy task, because these environments have other needs than modern systems. De Schutter [45] has made an explicit account of the rationale behind and the design of an aspect language support for typical legacy (Cobol) systems. Other researchers have discussed specific facets of aspect language design in legacy environments [16,10]. From this work, we have distilled five requirements for aspect languages for the re-engineering of legacy systems:

Base integration. The aspect language constructs should blend with the base programming language.

Expressive pointcuts. The pointcut language has to make up for weaker base language provisions regarding typing, structuring, etc.

Generic advice. Advice should be robust to small variations in types and context across the advised join points.

Join point context. Advice should have access to join point context.

Available weaver. A solid weaver should be available.

The first requirement considers the psychological integration of a new technology. As Cobol programmers are fluent in writing Cobol code and mostly weary of new technologies, adoption of aspects can be accelerated if the aspect language does not try to copy or re-implement existing features [25] and is suited to the particular domain programmers are working in. Instead of a separate `aspect` construct as in AspectJ, it is much more natural to adopt ordinary Cobol files as aspects. New constructs for pointcuts and advice have to be added, but to lower the learning curve they should be as close as possible to existing language constructs (C preprocessor, etc.) and should be able to interact with them. This also makes integration into existing development environments easier, because these only need to support the new advice and pointcut concepts.

The second and third requirement can be illustrated best by an example. Bruntink et al. [9,10] have used AspectC [16], the first aspect language for C (described later), to implement an aspect which checks whether or not pointer arguments passed to a procedure correspond to a `null` pointer. As C does not have a kind of "super-type" similar to Java's `Object`[7] and it does not support C++-like templates, there is no type-safe way to refer to a generic type. Because AspectC does not have explicit provisions for dealing with this, Bruntink et al. [9,10] are forced to duplicate their argument checking advice for each occurring argument type and to use plain enumerations of procedure names as pointcut. This situation impeded maintenance, as the long enumeration-based pointcuts had to be adapted on every non-trivial source code change, and changes to the advice logic had to be percolated to all duplicates of the advice.

To resolve these problems, Bruntink et al. [9,10] have developed a domain-specific language (DSL) for parameter checking, which is translated by a preprocessor to AspectC advice. Although they show that their solution greatly improves the source code quality, it still remains an ad hoc solution. Aspect languages for legacy systems should provide support for writing robust pointcuts and to specify generic advice, i.e. advice which is robust to small variability in types and context across all join points it advises. Robustness can further be improved by providing more advanced join points (variable access, control flow, etc.), whereas the ease of expressing the precedence between aspects or individual aspects are also important to keep in mind for genericity.

The fourth requirement, i.e. sophisticated access to join point context, refers to the base elements in terms of which pointcuts are expressed. To be able to specify robust patterns of join points, Gybels et al. [34] and De Schutter [65] have proposed access to program structure as a prerequisite. De Schutter has elaborated on this by stressing the importance of weave-time meta data in

[7] C does have `void` pointers, which can point to anything, but using them precludes compile-time type checking.

pointcuts, i.e. logic facts which represent design information or results of reverse engineering analysis. They allow to write more robust pointcuts which is synchronised with design changes, more precise analysis results or e.g. developer annotations. In general, any kind of information could be offered as context to pointcuts [57], or directly to advice. The latter is typically obtained by means of a specific join point object (e.g. named "thisJoinPoint").

The fifth requirement seems trivial, but for many aspect languages only a proof-of-concept implementation exists, which is not able to cope with the actual code found in legacy systems. Robustness to base language dialects and the ability to deal with language abuse (function pointers e.g.) are indispensable. The moment in time on which the weaver kicks in can be important. Many aspect languages for C feature a compile-time weaver, primarily because the typical domains where C shines (system software!) require highly efficient woven code. However, these systems have other desirable properties too, such as availability and debuggability. These are the application areas run-time weavers can be beneficial [59] for, as they theoretically offer the capability to advise any running system. For this, most dynamic weavers are based on instrumentation libraries or techniques like code splicing [27], i.e., tweaking the assembler code to jump to aspect code. Consequently, most of them do not require to parse or process the actual source code. Platform-independence of the instrumentation mechanisms in use is questionable, however. Also, there usually is no opportunity to optimize the advised application after the dynamic weaving: they tend to become patchworks. This is not the case for static weavers.

To summarise, aspect languages for legacy systems should blend naturally with the base programming language, should support generic advice, offer a means for composing expressive pointcuts and access to join point context. A sufficiently mature weaver implementation is also needed. Any language which fits these requirements will also provide us with two of our three needs for making dynamic analysis work (cfr. Section 2.3). *[Reviewer 1 says: remove reference to dynamic analysis?]*

At the start of our experiment we had a choice of four AOP tools for C: AspectC, AspectC++, AspectX and Arachne. We have evaluated these four tools with respect to the five requirements. The following sections discuss in more detail Table 1, which summarises our results.

3.3 AspectC

AspectC has been the first aspect language for C, inspired by AspectJ's constructs. Figure 2 is only interested in (lines 1–3) the execution of the

	AspectC	AspectC++	AspectX	Arachne	Aspicere
domain	kernel	general	general	systems	general
base integration	+	.h	XML	+	+
preprocessor	-	-	#include	-	-
PCD robustness	-	regexp	XPath	+	LMP
(function) pointers	-	-	-	-	-
ITD	-	+	+	-	-
basic join points	AspectJ	AspectJ	+	+	+
dynamic join points	AspectJ	AspectJ	-	+	-
advanced join points	-	`callsto/reachable`	comments	+	-
variable access	-	-	+	+	-
generic advice	-	+	+	-	+
aspect interaction	build	explicit	build	deployment	build
advice interaction	lexical	lexical	lexical	lexical	lexical
context	-	+	+	+	+
thisJoinPoint	-	+	-	-	+
annotations	-	-	-	-	-
availability	+	+	+	+	+
weaver type	source2source	source2source	source2source	run-time	source
optimisation	-	+	-	-	-
K&R support	+	-	+	N/A	+
IDE support	-	+	-	-	-

Table 1
Overview of existing aspect languages for C. Each set of rows corresponds to the provisions for one of the five requirements for aspect languages for legacy systems. A +/- indicates good/bad support for a feature, whereas "N/A" signals when an entry is not applicable to a language. Because every aspect language supports access to function arguments and global variables, a "-" for "context" means that there is no additional means for access to context.

```
1 pointcut allocating_buffers(vm_object_t obj, vm_pindex_t pindex):
2  execution(vm_page_t vm_page_lookup(obj, pindex))
3  && cflow(execution(int allocbuf(struct buf*, int)));
4
5 around(vm_object_t obj, vm_pindex_t pindex):
6  allocating_buffers(obj, pindex){
7    vm_page_t m = proceed(obj, pindex);
8    if ((m != NULL) && !(m->flags & PG_BUSY)
9       && ((m->queue - m->pc) == PQ_CACHE)
10      && (pages_available() < vfs_page_threshold()))
11     pagedaemon_wakeup();
12   return m;
13 }
```

Fig. 2. AspectC aspect for page daemon wake-up in the FreeBSD kernel [15].

`vm_page_lookup` procedure if this join point lies in the control flow of an execution join point of the `allocbuf` procedure. Instead of each of these join points (lines 5–6), the advice body on lines 7–12 is executed. This is normal C logic, except for the call to `proceed` which enables to execute the advised

```
1 aspect ThrowWin32Errors{
2
3   pointcut Win32API() =  "% CreateWindow%(...)"
4                        || "% BeginPaint(...)"
5                        || "% CreateFile%(...)"
6                        || ...
7   ...
8   advice call(Win32API()): after () {
9    if(win32::IsErrorResult(*tjp->result())){
10     ostringstream os;
11     DWORD code=GetLastError();
12
13     os << "WIN32 ERROR " << code << " : "
14        << win32::GetErrorText(code) << endl;
15     os << "WHILE CALLING: "
16        << tjp->signature( << endl;
17     os << "WITH: " << "(";
18
19     // Generate joinpoint-specific sequence of
20     // operations to stream all argument values
21     stream_params<JoinPoint,JoinPoint::ARGS>::process(os,tjp);
22     os << ")";
23     throw win32::Exception(os.str(),code);
24    }
25   }
26 }
```

Fig. 3. AspectC++ aspect which converts return value error codes into C++ exceptions [71].

join point from inside the advice. This is specific to `around` advice, as `before` and `after` advice just execute before or after the actual join point is executed and cannot control the latter. There is no explicit `aspect` construct, as file boundaries are used for this. Contrary to AspectJ, variable accesses cannot be advised and there is no ITD support either.

The advice signature does not specify a return type (line 5). Instead, the aspect developer should determine the right return type when it is needed in the advice body, e.g. to declare local variables (line 7). Regular expressions cannot be used either in pointcuts, which means that for every possible return type a separate pointcut *and* advice has to be written. This has caused the problems of Bruntink et al. discussed in Section 3.2. Access to typed context is possible (line 5). Initially [16], aspects were hand-compiled, but later on [15], a real weaver has been built. AspectC seems unmaintained since 2003 without any official releases, but a weaver prototype has been available on request.

3.4 AspectC++

AspectC++ [70,71] [8] is the most mature and general-purpose aspect language for C++ to date, but since its inception people have tried to use it for C too.

[8] http://www.aspectc.org/

Official support for this has never been a priority, however. Non-ANSI C code (so-called "K&R"-code) cannot be parsed by the weaver. It is also not clear which constructs and pointcuts can be used for C and which ones cannot. The AspectC++ weaver is heavily based on template instantiation to reduce memory footprint and execution time. Hence, the woven code is valid C++ which needs a modern C++ compiler. There is a (commercial) IDE plugin.

As Figure 3 shows, AspectC++ is heavily influenced by AspectJ. There is an explicit `aspect` construct which is similar to a C++ class, hence it needs to be declared inside a special "aspect header file". Join point, advice and pointcut types are comparable to AspectJ. Contrary to AspectJ, advice and inter-type declarations ("slices") can be specified in the same way. Because of this, the join point model is said to be "unified". Join points are implicitly typed, such that the weaver may check that they are only advised by correctly typed advice. Regular expressions can be used to specify pointcuts (lines 3–6 on Figure 3). Two new pointcut types are provided. The `callsto` pointcut takes an `execution` pointcut and deduces which call join points can call the join points described by the `execution` pointcut. The `reachable` pointcut is analogous, but it calculates (via static analysis) all join points from which its argument join points can be reached. There are no `set` and `get` pointcuts, i.e. access to variables is not reified as a join point, because of aliasing problems introduced by pointers and because of the unsound semantics of `set` regarding `operator=`. Just like AspectJ, there are precedence directives to derive a total order between aspects. Advice ordering within an aspect is ordered via lexical conventions.

AspectC++ has coined the term "generic advice" [50] to refer to the powerful capabilities of templates for obtaining highly reusable and robust advice. The idea is that AspectJ's distinction between static and dynamic context provided by a `thisJoinPoint`-like construct is generalised to C++'s strong compile-time templating mechanism. Compile-time context can be used to instantiate templated advice and functions, such that there is no run-time overhead to dynamically allocate or access context. Line 21 of Figure 3 gives an example of this. Because `JoinPoint` is just a class name and `JoinPoint::ARGS` statically resolves to the correct number of function arguments of the advised call join point, the `stream_params<JoinPoint,JoinPoint::ARGS>` template is instantiated at compile-time using template meta-programming. This mechanism allows for very reusable and robust advice, which varies automatically based on the particular join point and advice context. As a downside, the templates can easily get very complex to understand, especially for C programmers which are used to the simpler semantics of the C preprocessor.

3.5 AspectX/XWeaver

```
1 <pointcut name="targetFloatNameGetter" type="src:name"
2           constraint="(text()='_flt') and
3            (not(contains(following-sibling::text()[1],'=')) or
4            (contains(following-sibling::text()[1],'==')))">
5   <restriction type="within">
6     <pointcutRef ref="anySampleClassExpr" type="src:expr" />
7   </restriction>
8 </pointcut>
9
10 <advice name="targetFloatNameGetter" type="replace">
11   <pointcutRef ref="targetFloatNameGetter" aspect="PointcutLibrary"
12                type="src:name" />
13   <codeModifier type="codeFragment">
14     <xsl>pDB-&gt;getParameterFloat(PD<xsl:value-of
15          select="upper-case(current())" />)</xsl>
16   </codeModifier>
17 </advice>
```

Fig. 4. Accesses to a float member are replaced by the result of a method call with XWeaver (see example from website).

XWeaver [9] [63] is the name of the aspect weaver associated with the AspectX aspect language. It is conceived for tailoring software frameworks to control systems. As quality control is important for this, XWeaver's task is to generate woven code which syntactically resembles the base code layout and even updates comments (to document the woven code) such that the woven code can be manually investigated. XWeaver does not work on the program AST, but on srcML. This is an XML representation of a program in which only high-level program constructs are accessible. Comments and include/import statements are retained. This format makes XWeaver language-independent, in the sense that initial C++ support has been extended to Java once srcML was released for Java. Just as is the case with AspectC++, XWeaver can be used for C systems too.

The AspectX language is also XML-based, as the advice in Figure 4 shows. XML Schema is used to type-check the syntax of the XML aspects. AspectX allows the usage of XPath and XSLT technologies in the pointcut (lines 2–4) and advice (lines 14–15) respectively. XPath is able to select the right XML nodes by navigating across the XML tree. In the example, nodes of type `float` are selected (line 2) which do not occur as the left-hand side of an assignment or "equals" condition (lines 3–4). The advice of lines 10–17 replaces (line 10) the selected elements using the XSLT transformation of lines 14–15. This transformation capitalises the name of the advised join point XML node. Hence, the user should have considerable knowledge of the program XML-model. Special symbols need to be escaped, as the `>` on line 14 shows. Inclusion of XML documents can be exploited to reuse a library of pointcuts. To summarise, the AspectX language is a very low-level aspect language which resides on the border with pure program transformation.

[9] `http://www.xweaver.org`

```
1 seq( call(void* malloc(size t)) && args(allocatedSize)
2                                 && return(buffer);
3      write(buffer) && size(writtenSize)
4                    && if(writtenSize > allocatedSize)
5       then reportOverflow();
6      call(void free(void )) && args(buffer); )
```

Fig. 5. Buffer overflow detection aspect [22].

Join point context (argument types/names, return types, etc.) is accessible via dollar-variables like `${className}`. Under the hood, XWeaver transforms aspects in XSLT transformations, which means that join point context actually corresponds to XSLT queries. Hence, users can extend XWeaver with new context queries. New join point types can be added in a similar manner, and they can be very fine-grained like e.g. if-blocks, return-statements, etc. More traditional join points like `execution` exist, but all of them are purely statically determined based on the AST. There are no provisions for dynamic join points. On the other hand, the focus on program transformation enables syntactic ITD of comments and even include/import statements.

XWeaver is implemented in Java. There is an Eclipse plugin (AXDT) akin to the AspectJ AJDT, but command line or build script access (via Ant) is also possible. XWeaver can generate an Ant file based on a project file (XML). The latter specifies the important directories in the project and also the aspect configuration per subset of base code modules and header files.

3.6 Arachne

Arachne [10] improves on the μDiner framework [22]. The pointcut language has been reworked, inspired by Prolog. The sequence of nested calls has been generalised to an arbitrary sequence of procedure call and/or (in)direct variable access join points, whether or not they are nested. The resulting `sequence` pointcut is a natural means for advising protocol-like behaviour, as each element of the sequence can be advised individually. The advice of Figure 5 detects when more data is written into a heap-allocated buffer (lines 3–4) than initially allocated (lines 1–2). In that case, overflow is reported (line 5). The sequence ends when the buffer is deallocated (line 6). The latter is required to avoid that further run-time checks are performed for the buffer allocated at that address.

To increase the expressivity of this language, Loriant et al. [51] later have added the possibility to bind specific context to each instance of a `sequence`. Also, a `fakeEvent` construct has been introduced to simulate calls to an arbitrary procedure when some join point matches. These fake events can then

[10] `http://www.emn.fr/x-info/arachne/`

trigger other advice of which the pointcut is expressed in terms of that event.

Clever assembler manipulation techniques are used to instrument a running system without having to pause it. Despite claims of robustness across computer architectures, these techniques did not work on the various test machines we have tested it on [11] . The last public release of Arachne dates back to March 2005.

3.7 Discussion

First, we have observed that most aspect languages blend well with the base code. Second, regular expressions are the most widespread mechanism to obtain expressive pointcuts. AspectX and Arachne are more powerful because of their syntactic program transformation and `sequence` pointcuts respectively. Third, except for AspectC++, the aspect languages provide some form of generic advice especially by allowing access to a rich set of join point variables in various ways inside the advice body. AspectC++ on the other hand relies on C++ templates. It does not require developers to change the weaver implementation to add extra context for obtaining more expressive pointcuts. Fourth, aspect languages with generic advice all have access to a wealth of join point context. Fifth, AspectC and AspectX have sufficiently robust weavers, whereas AspectC++ generates more efficient woven code. Arachne does not need to parse the base code, but yields less optimised woven programs. Overall, AspectC++, AspectX and Arachne conceptually are the best aspect languages for C based on our five requirements.

Because AspectC++ does not fully support C (e.g. no K&R parser) and generates C++ code instead of C, AspectX reasons in terms of XML transformation instead of in terms of join points, and Arachne does not provide generic advice and is not platform-independent, none of the languages are really suited for the legacy system environment we are targeting. Instead, we have decided to design and implement a new aspect language for C, i.e. *Aspicere*. The next section presents the design of *Aspicere*.

4 Aspicere, an AOP language and tool for legacy C systems

This section presents our aspect language for C, *Aspicere* [12] [79]. We consider its rationale, the language itself and the weaver we have developed for it. The

[11] Arachne is distributed as a live Linux distribution.

[12] Aspicere: verb, Latin, "to look at". Root of our modern word *aspect*. Available at `http://users.ugent.be/~badams/aspicere/`

last column of Table 1 shows how Aspicere compares to the aspect languages for C discussed in the previous section.

4.1 The join point model

As with the other aspect languages for C, procedures are the prime join point type in *Aspicere*. Similar to AspectJ, a distinction is made between a `call` and an `execution` join point, i.e., a join point at the caller and callee side. This helps to distinguish between advising all (`execution`) or just a number (`call`) of procedure invocations, which is important, e.g., regarding shipping aspects with libraries or not. In addition, an `execution` join point is the easiest way of dealing with function pointers.

Second, pointers also cause the problem of "aliasing". A given (global) variable can be accessed directly or via some pointer to it. Arachne (Section 3.6), e.g., tried to use the operating system's page fault mechanism to detect variable access, but this caused extreme performance penalties. Just like AspectC++ (Section 3.4), *Aspicere* does not support variable access join points like `get` or `set`.

Finally, *Aspicere* does not take inline assembler and preprocessor constructs like macros or conditional compilation into account.

4.2 Pointcuts

As argued in [65], legacy languages like C lack sufficient structure or reflective capabilities to be able to write crisp and robust pointcut patterns and advice. To deal with this, *Aspicere*'s pointcut language is heavily influenced by the querying variant of logic meta-programming (LMP) [76,6,34]. The basic idea is that a program is represented as a collection of logic facts and that a Turing-complete logic language is used to reason about these program facts. Pointcuts can be expressed in terms of the program facts to compose powerful patterns based on program structure using conjunction, disjunction and or negation (by unprovability). By carefully designing more advanced predicates in terms of more primitive ones, a clean, layered pointcut language can be constructed. Adding new pointcuts comes down to defining new logic predicates. By raising the level of abstraction, pointcut predicates can be brought closer to the actual problem domain.

Lines 2–4 of Figure 6 show an example pointcut in *Aspicere*. This pointcut

```
1 ReturnType safe_ato(TYPE ReturnType,char* Src) around Jp:
2   invocation(Jp,"ato.")
3   && args(Jp,[Src]))
4   && type(Jp,ReturnType){
5     ReturnType dst;
6
7     if(Src == NULL) dst = 0; /* compiler does the cast */
8     else dst=proceed();
9
10    return dst;
11 }
```

Fig. 6. Aspect to make conversion to numbers null pointer-proof.

matches all calls[13] (line 2) to procedures of which the name matches the regular expression "`ato.`", i.e., the name starts with `ato` and the fourth letter is arbitrary. Because *Aspicere* does not allow advising a call via a function pointer (see Section 4.1), this pointcut only matches explicit function calls. To connect the `invocation` predicate to the ones on lines 3 and 4, Prolog's unification allows us to reuse the previously bound join point variable (`Jp`). This is actually a very natural way to express that two bound variables should be equal. The `args` predicate binds the sole argument passed via the argument list to the `Src` variable and also captures the procedure call's return type as `ReturnType`.

The unification has an interesting effect: if a variable can have multiple values, each one is eventually used to find a complete match of the logic rule. In Figure 6, each function call which satisfies the regular expression on line 2 (and the other conditions) leads to an extra match of the pointcut.

Apart from program structure, logic facts can represent weave-time meta data. This is especially useful for legacy systems, because meta data facts can record information obtained via reverse engineering and make it accessible in other advice to re-engineer the system. Design information, the actual composition of source modules (is one executable built or are multiple libraries built?), information of base code modules selected by the user, etc. can all be passed to pointcuts and be used in the advice body. Logic facts are useful to store meta data separately in a loosely coupled fashion.

To summarise, *Aspicere* has an expressive pointcut which is able to abstract over implementation details of the base code. The binding of variables enables access to a variety of join point context (more on this in the next section).

4.3 Advice construct

Aspicere's aspects correspond to ordinary C modules with a new advice con-

[13] Because of name clash issues in our weaver implementations, we use `invocation` instead of `call`.

struct. Figure 6 shows an example advice. It secures calls to the standard `atoi`, `atol` and `atof` procedures to prevent a program from crashing [14] when a null pointer is passed as an argument. These three procedures should parse a string (`char*`) argument into an `int`, `long` or `double`. Note that a single advice suffices to advise all three procedures because of the use of so-called "template parameters" (lines 1 and 5), which are similar to C++ templates.

An advice structure specifies:

- the advice return type (useless in case of `before`- or `after`-advice);
- the name of the advice;
- a list of bound context variables [15] visible to the advice body;
- the type of advice (`before`, `around` or `after (returning)`);
- (in case of `after returning`) binding of return value to a variable;
- the name of the joinpoint variable where advice has to be woven;
- a pointcut (behind the colon);
- the advice body.

Aspicere's advice body contains pure C code enhanced with template parameters. As the pointcut (see Section 4.2) consists of Prolog predicates with C-like conjunction, disjunction and negation operators, this makes *Aspicere* a hybrid of pure C and a Prolog-based pointcut language. To make the learning curve lower, *Aspicere* enables a C-like syntax of Prolog's conjunction, disjunction and negation operators (e.g., "`&&`" instead of "`,`").

The advice of Figure 6 corresponds to `around`-advice on join points `Jp` which satisfy the pointcut on lines 2–4. This pointcut has been explained in the previous section. Two typed variables are bound (`ReturnType` and `Src`) and are available for use as template parameters in the advice body. `Src` is a simple string, whereas `ReturnType` represents an actual C `TYPE`. This is a custom (meta)type we have added to *Aspicere*, because C does not have reflective capabilities. One can use such a type parameter further on in the binding list, as return type of the advice (line 1) and of course inside the advice body itself (line 5). Type parameters help to achieve better static typing than the use of a catch-all `void*` would allow, similar to C++ templates. Advice becomes robust to small changes in, e.g., types, as Figure 6 illustrates.

Apart from template parameters, an advice body may contain a `proceed`-call, similar to AspectJ. If no arguments are given, the join point's original arguments are passed as is to any remaining advices on the same join point or to the join point itself. If one wants to replace the value of the arguments, one should fill in the arguments or assign directly to a bound argument. Another

[14] Some platforms (e.g., UnixWare) already handle this, others (e.g., GNU) do not. The advice shown here allows to abstract away from this difference in platform.
[15] Their names always start with a capital letter.

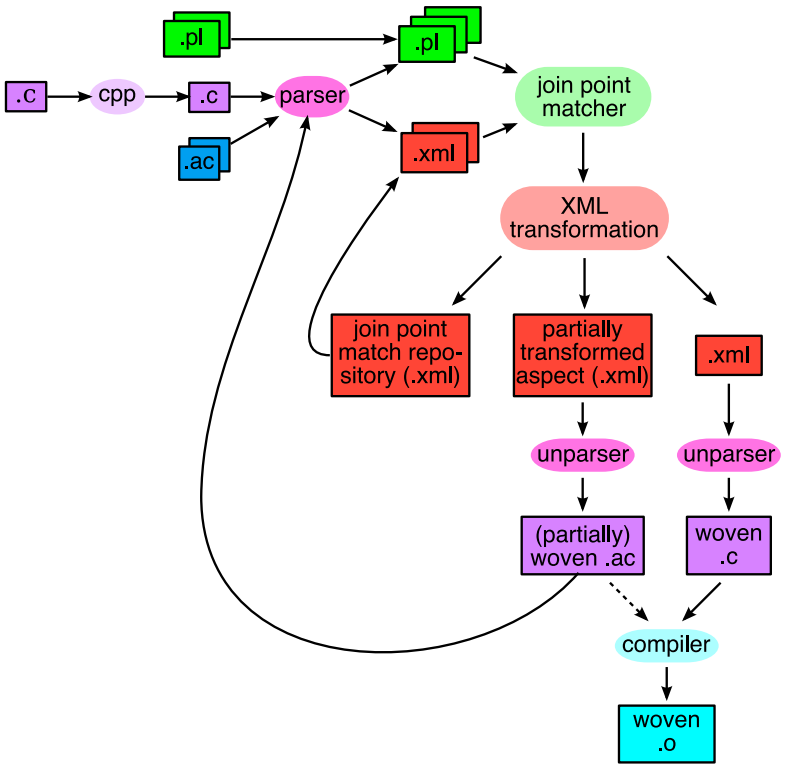


Fig. 7. Architecture of Aspicere1, *Aspicere*'s source-to-source weaver. The .ac-files represent aspects, while .pl-files contain logic predicates.

alternative is to access the thisJoinPoint-like struct, which is accessible via the join point variable (`Jp`). This `struct` contains the following fields:

nrArgs — number of arguments
args — array of arguments
returnValue — pointer to return value
fileName — name of file in which advised join point resides
functionName — name of advised function

Aspicere provides generic advice with flexible access to join point context. It forms a hybrid between C and Prolog. As `before`-, `after`- and `around`-advice can be used, the third requirement of Section 2.3 is satisfied as well.

4.4 Aspicere's weaver

Figure 7 shows the architecture of the *Aspicere* weaver (named "Aspicere1"). It is a pure source-to-source weaver [79], which takes as input base code (one module at a time), aspects (.ac) and Prolog modules (.pl), and generates woven C code which can be compiled using the normal compiler. This means that the weaver has to be integrated between the C preprocessor and the C compiler.

Aspicere's parser is capable of handling C code made up of a mix of K&R-, ANSI- and GNU-standards.

Inside the weaver, a parser and unparser convert the C code to/from an XML representation of the AST (analogous to AspectX in Section 3.5). The Prolog modules and the pointcuts (transformed into Prolog rules) are used to locate the right join point shadows[16] [37], i.e., the appropriate XML nodes of the AST. Once these have been found, the XML tree representing the base code module can be transformed, as well as the aspects themselves.

Advice is converted into multiple C functions, one per combination of type parameters used in the advice body. These transformed advices are collected into the transformed aspect, which has to be reused as input to the weaving process of other files of the same application (the big loop in Figure 7). At the same time, information about matched join points is collected into a "join point match repository" XML file, which is also reused in future weavings. The woven base code module is converted back to C code, whereas the transformed aspect possibly needs to be transformed further when weaving other base code modules. Eventually, the resulting transformed aspect also has to be compiled and linked with the base code to form the complete woven system.

Aspicere fulfils the five requirements for aspect languages for legacy systems set out in Section 3.2, and hence implicitly also two of the three requirements for supporting dynamic analysis (cfr. Section 2.3). The next section shows how this is applied to the experiment, and fills in the last requirement (the execution scenario).*[Reviewer 1 says: remove reference to dynamic analysis?]*

5 The experiment

[Reviewer 1 says: rename+move first subsection to section 2] It is now time to discuss the actual experiment and its results. As was discussed in the previous sections we apply dynamic analysis using an aspect written in *Aspicere* for collecting the trace.

5.1 The setting

Kava, our industrial partner for this experiment, is a non-profit organization that groups over a thousand Flemish pharmacists. While originally set up as a

[16] A join point is a run-time concept. However, for each join point a corresponding location in the source code can be found which contains the actual code which is executed by the join point (e.g. an actual procedure call forms the shadow of a `call` join point). Shadows are used by compile-time weavers to statically weave aspects.

union for the pharmaceutical profession, they have evolved into a full-fledged service-oriented provider for pharmacists.

Some ten years ago they have developed a suite of applications written in non-ANSI C, which has put them among the first in the industry to have an automated tarification service. Due to successive health care regulation changes they are very much aware of the necessity to adapt and re-engineer this service. Furthermore, during their migration from non-ANSI C to ANSI C compliant versions of their applications, they have noted that the documentation of these applications was outdated, making it difficult for new software engineers to get acquainted with the system. To help solve this problem, Kava was interested in applying reverse engineering techniques to their system. They thus presented us with an opportunity to apply the dynamic analysis techniques mentioned earlier.

The developers at Kava have pointed us to the so-called *TDFS*[17] batch application. They use this as a check to see whether adaptations in the system have any unforeseen consequences. As such, it should be considered as a functional application —it outputs a detailed invoice of all prescriptions, ready to be sent to the health care insurance institutions—, but also as a kind of regression test. As an added bonus the use of this scenario allows us to verify whether weaving the TDFS application is indeed behavior preserving in the sense that we compared the results generated by a vanilla version and a woven version of the Kava system. *[Kris says: diff Kava en TDFS] [Reviewer 1 says: other characteristics of TDFS really needed if diff is OK?]*

5.2 Instrumentation of the application

As seen in the pointcut of Figure 8 (lines 2–4), we advise individual procedure calls whose name does not end in "printf" or "scanf" (line 2). At each affected join point, we want to output the relevant associated context information. This is accessible in the advice body through the thisJoinPoint struct, bound to `Jp` (lines 10 and 16). More specialized context information can also be obtained and used through bindings like `ReturnType` (line 7).

Aspicere's around-advice then enables us to output call- and/or return-sequence information (lines 9–10 and 15–16), which lets us reconstruct the program call tree on which the dynamic analysis operates.

5.3 Results

[17] TDFS: "*T*ariferings*D*ienst *F*actuur en *S*tatistiek", or "Tarification Service for Invoices and Statistics".

```
1 ReturnType trace(TYPE ReturnType,char* FileStr) around Jp:
2   invocation(Jp, ``^(?!.*printf$|.*scanf$).*$'')
3   && type(Jp,ReturnType)
4   && !!is_void(ReturnType)
5   && trace_file(FileStr) {
6     FILE* fp=fopen(FileStr,"a");
7     ReturnType i;
8
9     fprintf (fp,"before ( %s in %s ) \n",
10             Jp->functionName,Jp->fileName); /* call sequence */
11    fflush(fp);
12
13    i = proceed (); /* continue normal control flow */
14
15    fprintf (fp,"after ( %s in %s ) \n",
16             Jp->functionName,Jp->fileName); /* return sequence */
17    fclose(fp);
18
19    return i;
20 }
```

Fig. 8. One of the two applied tracing aspect, i.e., the one aimed at non-void procedures.

#	Module	Hubiness	#	Module	Hubiness
1	e_tdfs_mut1.c	0.814941	9	csroutines.c	0
2	tdfs_mut1_form.c	0.45397	10	UW_strncpy.c	0
3	tdfs_bord.c	0.397726	11	td.ec	0
4	tdfs_mut2.c	0.164278	12	cache.c	0
5	tools.c	0.164278	13	decfties.c	0
6	io.c	0.12548	14	weglf.c	0
7	csrout.c	0.0321257	15	get_request.c	0
8	tarpargeg.c	0			

Table 2
Results of the webmining technique.

Performing the webmining analysis gives us the results as listed in Table 2. The results are ranked according to the *hubiness* values, found in the third and sixth column, from high to low. Hubiness values lie in the range $[0, 1]$. Some important facts that can be derived from Table 2 are:

- Module `e_tdfs_mut1.c` stands out with a high hubiness score.
- Only seven out of the 15 modules have a value greater than zero. Modules with a value of zero do not call other modules.
- The four modules that are specific to the TDFS application (as can be seen from their names) show up in the four highest ranked places.

5.4 Validation

We now cover the results that we have obtained from applying the key class identification technique to the subject software system. For the specific module

that we have investigated, namely the TDFS module, two developers, D_1 and D_2, were responsible at Kava. Both have thorough and active knowledge of the structure and the inner workings of this particular application. As such, we also present the feedback these two developers had when presenting them our results.

Through our experiment, we have established that the TDFS application consists of 15 modules. We have presented the developers with a schema consisting of each of these modules, and have asked them the following three questions:

(1) Which module is the most essential?
(2) Which module tends to contain the most bugs?
(3) Which module is the hardest to debug?

We have noted their answers and also have asked if there were any particular reasons why they believed a certain module to be important, hard to debug or to contain bugs. Subsequently, we have presented our results to each of the two developers separately. Afterwards we have discussed the results with both of them and have highlighted the similarities and differences in their comments.

During our discussion with the developers, D_1 mentioned *#1* (that is, source file number 1 as ranked in Table 2) and *#4* as being the most essential modules for the TDFS application. *#6* and *#12* are technically also important, but are not specific to the TDFS application, as they are used by many other applications of the system. D_1 was surprised at the fact that *#12* was not catalogued as being more important. *#7* and *#9* are difficult to debug, but only minor details changed in these modules in the last 10 years. D_2 clearly ranks *#1* as being the most important and most complicated module: it contains most of the business logic. *#4* makes a summary of the operations carried out by *#1* and checks the results generated by it. *#2* is mainly responsible for interaction with the end-user, while *#3* is concerned with formatting the output. As such, the opinions of D_1 and D_2 are indeed very similar, and support our own results. Furthermore, all modules specific to this application are ranked at the very top.

A last remark on one of the drawbacks of the webmining technique: container classes or modules are often ranked very low, because of the fact that their export coupling is low [80] (i.e., they do not call many procedures in other modules). This fact partly explains why *#12*, a caching data-structure which was expected to rank higher according to D_1, is placed quite low.

To summarize, we can say that the key class identification technique also works in a non-object-oriented environment. For our case study, the technique ranked the most important modules at the very top and as such identified these top-ranked modules as being "important". Furthermore, the original developers who cooperated in our study confirmed that we have indeed retrieved the most

important modules, making this technique very suitable for helping novice developers find their way in large-scale software systems.

5.5 Threats to Validity

This section discusses threats to the validity of our approach and results. We consider construct, internal and external validity.

5.5.1 Construct Validity

Construct validity is concerned with the meaningfulness of the measurements, i.e. do we get the measurements we need to draw conclusions? As we need a trace of a concrete run of the TDFS system, the aspect of Figure 8 indeed does what we want it to. Even the pitfalls in the concrete realisation with Aspicere's weaver in the Kava case do not endanger this. We discuss this in Section 6.

5.5.2 Internal Validity

Internal validity refers to the existence of other plausible rival hypotheses to explain our findings. The developer feedback gives a valid explanation, although each developer has most knowledge of his or her own set of modules. Similarly, the exact meaning of "important" modules may depend on a developer's understanding of this term. Some may refer to the architecturally rich modules we target, whereas others may think at the low-level modules which do the ground work (authorities), such as container classes. This is why we believe that the webmining yields correct results.

5.5.3 External Validity

Can we generalise our results to other systems and projects? This is the first application of webmining on a procedural system. There are enough similarities between procedural programming and object orientation (OO) to believe that we can generalise the approach to procedural programming in general. Where OO has e.g. polymorphism, procedural programming has function pointers. However, we need more experiments to validate this claim, but this is outside the scope of this paper.

6 Obstacles when applying AOP in legacy E-type systems

Having discussed the results of the analysis of the experiment, we now shift our focus to our observations of obstacles encountered when applying AOP to a legacy system. We consider three obstacles: physical integration of Aspicere1 into the Kava build system, problems with providing developers with the right notion of modularity and issues caused by K&R C code.

6.1 Integration into the build process

The first step in adding AOP to a legacy software system is to extend its build system such that aspects get woven into the final product. As we use a preprocessor-style weaver, this implies changing the compile cycle to:

(1) Preprocess
(2) Weave
(3) Compile
(4) Link

An actual example of this modification from the experiment is shown in Figures 9 and 10.

With regards to the experiment, the Kava applications use `make` to automate the build process. Historically, all 269 makefiles have been hand-written by several developers, not always using the same coding-conventions. During a recent migration operation from UnixWare to Linux, a significant number of makefiles have been generated with the help of *automake*[18], but not all. This means that the structure of the makefiles remains heterogeneous, a common attribute of (legacy) systems. This heterogeneity makes it hard to fully automate the modification of the build system as depicted in Figures 9 and 10, because the existence of certain environment variables is not ensured, invocation of compilers can happen in various ways, etc. This becomes apparent when, as in the case of our experiment, embedded sql preprocessing needs to be done (see Figures 11 and 12). On line 4, e.g., the `INCLUDE` environment variable is assumed to point to the right location of header files, whereas the original invocation of `esql` on line 2 of Figure 11 has to be transformed into a corresponding invocation of the C compiler on line 11 of Figure 12. Context-specific makefile changes are required.

An alternative to these makefile changes would be to re-route $(CC) and $(ESQL) to custom shell scripts ("wrappers") which execute *Aspicere* in the right way before invocation of the real $(CC) and $(ESQL) compilers. This too is problematic as the heterogeneity of the makefiles does not guarantee

[18] `Automake` is a tool that automatically generates makefiles starting from configuration files. Each generated makefile complies to the GNU Makefile standards and coding style. See http://sources.redhat.com/automake/

```
$(CC) -c -o file.o file.c
```

Fig. 9. Original makefile.

```
$(CC) -E -o tempfile.c file.c
cp tempfile.c file.c
aspicere -i file.c -o file.c \
         -aspects aspects.lst
$(CC) -c -o file.o file.c
```

Fig. 10. Adapted makefile.

```
.ec.o:
   $(ESQL) -c $*.ec
   rm -f $*.c
```

Fig. 11. Original *esql* makefile.

```
.ec.o:
   $(ESQL) -e $*.ec
   chmod 777 *
   cp `ectoc.sh $*.ec` $*.ec
   $(ESQL) -nup $*.ec $(C_INCLUDE)
   chmod 777 *
   cp `ectoicp.sh $*.ec` $*.ec
   aspicere -verbose -i $*.ec -o \
            `ectoc.sh $*.ec` \
            -aspects aspects.lst
   $(CC) -c `ectoc.sh $*.ec`
   rm -f $*.c
```

Fig. 12. Adapted *esql* makefile.

that in all cases there is a direct use of these commands, which is a typical problem with wrappers. Even if the wrappers are applied consistently, some tools like the $(ESQL) compiler, e.g., internally invoke the original C compiler. As this one is replaced by a wrapper, Aspicere1 eventually would be executed twice instead of once. In the end, a wrapper approach is not feasible either.

Real tool support for adapting makefiles is needed, and is something we have been focusing on as a result of this experiment. MAKAO [19] [1] is a re(verse)-engineering framework for build systems which enables visualisation, querying, filtering, verification and re-engineering of the build dependency graph. The makefile re-engineering support is itself based on AOP and facilitates exploiting context information to develop robust makefile re-engineerings. Unfortunately, MAKAO was not around at the time of the case study. A regular expression transformer has been used instead, but the lack of context for the refactorings required us to manually verify and fix all changes.

6.2 Linking aspects into the system

Aspicere1's weaver conceptually transforms aspects into C compilation units. This involves transforming the advice constructs into C procedures (so-called "advice instances"). All advised procedure calls in the base code are replaced by (indirect) calls to the right advice instance. After the transformed aspect module is compiled, it should be linked with every object file in which advised join points reside. As a bonus, one can share static and global state between

[19] `http://users.ugent.be/~badams/makao/`

all advice instances, e.g., the file pointer to which we send the tracing data.

This linking is, again, problematic due to the complexity and heterogeneity of the build system. As the experiment precludes any knowledge of the legacy system, and as we had no tool support for automating modifications of the build system, the linking as described above was simply not feasible. More in particular, the mapping of source code components on build system units could not be determined. Conceptually, aspect developers reason in terms of one software system, whereas in the build system the software has been split across multiple libraries and executables. A static weaver like Aspicere1 needs to process these build components in such a way that the aspect developer's notion of modular reasoning is still valid [41,33]. As we did not know which build units were part of the build system and how they were mapped on the source code, we could not optimally exploit Aspicere1's provisions (transformed aspects) for safeguarding the source code modularity.

To circumvent this, we have inserted the advice instances of a particular aspect into each advised base module at a time (as needed). This prevents the linking problems, but this also means that one can no longer share data. This is the reason why each advice instance manages its own file pointer (see line 6 of Figure 8), which results in spurious opening, flushing and closing of the actual trace file. This is a clear example of a situation where limitations in the build system backfire at the source code.

In the meantime, we have developed a link-time weaver for Aspicere ("Aspicere2") [2], which operates on the compiled source code during linking. Applications which span across multiple libraries and executables still require knowledge of the build architecture, but because libraries and executables reside at a slightly higher level than object files, integration of Aspicere2 into a build system is easier. A tool like MAKAO [1] provides the means to discover a build system's architecture.

6.3 Coverage of non-ANSI C language features

As Kava's system is a mix of ANSI and K&R style C code, Aspicere1 has to take care that it covers deviations between these dialects. As an example, procedure declarations with an empty argument list are allowed in K&R style C, whereas they are not in ANSI C. Actual declaration of the arguments is then postponed to the corresponding procedure definitions. The type inferencing required to handle this in the weaver is rather complex, and was therefore not fully integrated into Aspicere1 by the start of the experiment. As a result, some join points were skipped, introducing some errors in our measurements. To be more precise, we have advised 367 files, of which 125 contained skipped

join points. Of the 57015 discovered join points, only 2362 were filtered out, or a minor 4 percent. Random screenings of the code found that calls to the same small group of procedures were responsible. These procedures turned out to be very low level, not part of the business logic, and therefore ignoring them did not impact the analysis.

6.4 Conclusion

To summarise, the application of aspects in the source code for reverse engineering and re-engineering of a legacy system entails more than just adding source code. The new tools, i.e., the aspect weaver, need to be integrated into the existing development environment. Even without the demand for fast incremental weaving or IDE support, this proves difficult. The major cause of these problems is the gap between the notion of modularity in the source code, and the one supported by legacy build systems. Bridging this gap is hampered by the lack of understanding of the build system. Tool support is needed to deal with this [1].

7 Related work

To the best of our knowledge there are no other large-scale industrial experience reports on the use of dynamic analysis. Instead, this related work section touches upon (1) industrial experiences with static analysis and (2) a number of dynamic analysis techniques.

7.1 Industrial experiences

Moise and Wong describe their experiences with extracting knowledge from C/C++ systems in [53]. They use *Rigi* as a fact extractor for the C/C++ source code and focus on providing different decomposition approaches of the system, trying to satisfy different understanding needs. Because of the large-scale industrial case study they are working on, they are very much concerned with the scalability of their tool-chain.

In [77], Wong et al. describe their experiences with re-documenting industrial legacy applications with the help of their *Rigi* static reverse engineering environment. They have applied Rigi on COBOL, C and PL/AS [20] systems. The PL/AS experiment described in [77] exhibits a close resemblance with our own

[20] Programming Language/Advanced Systems (IBM).

experiments, as the goals and setting were very similar: a large scale industrial legacy application with 2 MLOC and 1300 compilation units (here not in C, but in a proprietary language). Because of the large scale of the application, they have also focused on delivering scalable reverse engineering techniques. One of the most significant lessons they have learned from their experiments is that in-the-large design documents describing the architecture of the software system's current state can be very beneficial for building up understanding of a software system and maintaining it. As such their goal is very similar to ours.

Another study by Moise and Wong highlights the fact that reverse engineering Java, C, C++, C#, Cobol systems, etc. is sometimes not enough [54]. Often, (implicit) knowledge is present in programs that are written in a variety of "scripting languages", such as Perl. Sometimes these Perl programs are standalone, but often, these programs written in scripting languages are *gluing* together large-scale industrial systems, written in a variety of programming languages, making it worthwhile to also reverse engineer the knowledge contained in these scripting language-systems. This is in line with our problems with Kava's build scripts.

Software architecture recovery has been subject to a lot of research before, e.g., of the Linux kernel [5], Mozilla [29], etc. Most of them leverage static analysis to accomplish their task on a repository of facts extracted from source code, linked object files, etc. Combining various approaches has led, e.g., to the Portable BookShelf [28]. We believe that the results of our dynamic analyses are complimentary to these efforts and enhance them to get a more complete view on a (legacy) software system.

Other related work is that of Riva [61], who provides an industrial experience report of *reverse architecting* software for mobile phones, and Linos et al. [49], who concentrate on understanding multi-language program dependencies.

7.2 Dynamic analysis

There has recently been a renewed research interest in the area of reverse engineering with the help of dynamic analysis. Most of these recently presented techniques can be classified according to their area of focus. As such, we identify: (1) novel visualization techniques, (2) feature localization techniques and (3) smart abstraction techniques that reduce the amount of trace data before it is interpreted (or visualized). We now give a brief overview of recent work in each of the aforementioned categories.

[Reviewer 1 says: dynamic call graph extraction of Murphy et al. from 1998?]

Visualization. *[Reviewer 3 says: redundant?]* Visualization is often used as a way to cope with the huge amounts of data that go with dynamic analysis. The visualization strategies range from visualizations similar to UML sequence diagrams (e.g., De Pauw et al. [58]) to more elaborate visualization strategies. One of these is the hierarchical bundling visualization, as presented by Cornelissen et al. [19]. Greevy et al. present a technique that shows running software in 3D [32]: their visualization uses the metaphor of constructing buildings, whereby objects in the software that are very active are depicted as 'higher' buildings. Ducasse et al. on the other hand use polymetric views to condense run-time information [24].

Feature localization. Feature localization is concerned with identifying those parts of source code that are responsible for executing a feature, whereby a feature is defined as a unit of human-observable computer-action. In this context Eisenbarth et al. use formal concept analysis to correlate features with source code [26], while Greevy et al use two complementary perspectives to correlate features to source code and vice versa. In [43] Kuhn and Greevy exploit the analogy between traces and signal processing to visualize traces and identify (common) features in a set of traces.

Abstraction techniques. Hamou-Lhadj et al. have presented a technique that detects (and eliminates) low-level (function) calls from traces [36]; their technique allows to focus on the more high-level behavior of the application under study. In other work Hamou-Lhadj and Lethbridge propose to apply automatic text summarization techniques to execution traces, again to reduce the trace size and focus on high-level behavior when trying to understand a piece of software [35]. Also work of Zaidman proposes to abstract traces using a heuristic based on the relative frequency of execution of events in an execution trace [81].

To the best of our knowledge none of these recently presented dynamic analysis approaches have been tried out in a large-scale industrial legacy environment, but rather only on academic examples and small to medium scale open source software projects. For a more detailed overview of work in the field of dynamic analysis, we refer to [78,30].

8 Conclusion

This paper reports on the application of dynamic analysis with the help of aspect-oriented programming (AOP) for reverse engineering a legacy E-type system.

Using dynamic analysis allowed us to follow a goal-oriented strategy, i.e., it allowed us to analyze only specific parts of the system, which is a benefit when working with large-scale applications. The analysis technique of choice, the key module detection technique, enabled us to identify those modules that should be investigated first when trying to understand the application. We validated that our solution does indeed identify those need-to-be-understood modules, with the input of the original developers.

AOP was used to collect the necessary trace information in a modular and non-invasive way. For deciding which AOP solution to use, we have first undertaken a survey of existing AOP languages and tools for C. Ultimately none of the surveyed tools fit our re-engineering requirements, leading us to construct our own AOP solution for C, named *Aspicere*. *Aspicere* targets legacy environments, has an expressive pointcut language based on logic meta programming (LMP) and features generic advice with access to join point meta data.

While the solution proved to work very well and is conceptually very clean, it comes with a major quid pro quo, namely that the integration of an aspect solution in a legacy build environment can prove troublesome. A first reason for this is the fact that the modular reasoning of traditional build systems conflicts with the non-hierarchical nature of aspect-orientation. Secondly, the build system proved to be legacy itself, which made it difficult to automate any adaptation of it; indicating that the build system itself is also in need of a re-engineering operation.

Overall, we can say that the dynamic analysis proved useful for detecting the most important modules to be used during early program comprehension. AOP enabled us to obtain the necessary trace information in a clean way. Yet, when integrating AOP into the build system of the legacy system, we found that the build system itself could benefit from a re-engineering step. We expect this situation to be common in legacy environments.

Acknowledgements

We would like to thank Kava for their cooperation and very generous support. Kris De Schutter and Andy Zaidman received support within the Belgian research project ARRIBA, sponsored by the IWT, Flanders. Kris De Schutter also received support from the Belgian research project AspectLab, sponsored by the IWT, Flanders. Bram Adams is supported by a BOF grant from Ghent University. Further support came from the Dutch NWO Jacquard Reconstructor and the MoVES project (Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy).

References

[1] Bram Adams, Kris De Schutter, Herman Tromp, and Wolfgang De Meuter. Design recovery and maintenance of build systems. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 114–123, Washington, DC, US, 2007. IEEE Computer Society.

[2] Bram Adams and Kris De Schutter. An aspect for idiom-based exception handling (using local continuation join points, join point properties, annotations and type parameters). In *Proceedings of the Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), co-located with AOSD*, 2007.

[3] Keith H. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[4] Alan W. Biermann and Jerome A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.

[5] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 555–563, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[6] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, pages 110–127, London, UK, 2002. Springer-Verlag.

[7] Michael Brodie and Michael Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach.* Morgan Kaufmann, San Francisco, CA, USA, 1995.

[8] Magiel Bruntink, Arie van Deursen, Maja D'Hondt, and Tom Tourwé. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 199–211, New York, NY, USA, 2007.

[9] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. An initial experiment in reverse engineering aspects. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 306–307, TODO, 2004. IEEE Computer Society.

[10] Magiel Bruntink, Arie van Deursen, and Tom Tourwe. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 37–46, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *Proceeding of the 28th international conference on Software engineering (ICSE)*, pages 242–251, Shanghai, China, 2006. ACM Press.

[12] Magiel Bruntink, Arie van Deursen, Tom Tourwé, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pages 200–209, Chicago, IL, USA, September 2004.

[13] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, TODO, 2004. TODO.

[14] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[15] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the International Conference on Aspect Oriented Software Development Conference (AOSD)*, pages 50–59, New York, NY, USA, 2003. ACM Press.

[16] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Software Engineering Notes*, 26(5):88–98, 2001.

[17] Adrian Colyer and Andrew Clement. Large-scale aosd for middleware. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65, New York, NY, USA, 2004. ACM Press.

[18] Thomas A. Corbi. Program understanding: Challenge for the 90s. *IBM Systems Journal*, 28(2):294–306, 1990.

[19] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 49–58, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Carlos Montes de Oca and Doris L. Carver. Identification of data cohesive subsystems using data mining techniques. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 16–23, Washington, DC, USA, 1998. IEEE Computer Society.

[21] Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, San Francisco, CA, USA, 2003.

[22] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the International Conference on Aspect Oriented Software Development(AOSD)*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[23] Stéphane Ducasse, Tudor Girba, and Roel Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 37–46, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 309–318, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Pascal Durr, Gurcan Gulesir, Lodewijk Bergmans, Mehmet Aksit, and Remco van Engelen. Applying AOP in an industrial context. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*, March 2006.

[26] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[27] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the International Conference on Aspect Oriented Software Development Conference (AOSD)*, pages 51–62, New York, NY, USA, 2005. ACM Press.

[28] Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong. Advances in software engineering. In Hakan Erdogmus and Oryal Tanir, editors, *Advances in software engineering*, chapter The software bookshelf, pages 295–339. Springer, TODO, 2002.

[29] Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Symposium on Constructing Software Engineering Tools (CoSET)*, 2000.

[30] Orla Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, Switzerland, 2007.

[31] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 314–323, Washington, DC, USA, 2005. IEEE Computer Society.

[32] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3d. In *Proceedings of the Symposium on Software Visualization (SOFTVIS)*, pages 47–56, TODO, 2006. ACM Press.

[33] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[34] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the International Conference on Aspect Oriented Software Development (AOSD)*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[35] A. Hamou-Lhadj and T. C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[36] Abdelwahab Hamou-Lhadj, Edna Braun, Danien Amyot, and Timothy Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 112–121, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[38] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[39] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[40] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer, 1997.

[41] Gregor Kiczales and Mira Mezini. Aspect-Oriented Programming and modular reasoning. In *International Conference on Software Engineering (ICSE)*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[42] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM Press*, 46(5):604–632, 1999.

[43] Adrian Kuhn and Orla Greevy. Exploiting the analogy between traces and signal processing. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 320–329, Washington, DC, USA, 2006. IEEE Computer Society.

[44] Ralf Lämmel and Kris De Schutter. What does aspect-oriented programming mean to Cobol? In *Proceedings of International Conference on Aspect-Oriented Software Development (AOSD)*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[45] Ralf Lämmel and Kris De Schutter. What does Aspect-Oriented Programming mean to Cobol? In *Aspect Oriented Software Development Conference (AOSD)*, pages 99–110. ACM Press, 2005.

[46] Meir M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[47] Meir M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, 1998.

[48] Meir M. Lehman and Laszlo A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., TODO and is this written by or edited by Lehman and Belady?, 1985.

[49] Panagiotis K. Linos, Zhi hong Chen, Seth Berrier, and Brian O'Rourke. A tool for understanding multi-language program dependencies. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 64–72, Washington, DC, USA, 2003. IEEE Computer Society.

[50] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Proc. Generative Programming and Component Engineering: Third International Conference*, volume 3286 of *svlncs*, pages 55–74. Springer, October 2004.

[51] Nicolas Loriant, Marc Ségura-Devillechaise, Thomas Fritz, and Jean-Marc Menaud. A reflexive extension to Arachne's aspect language. In *Proceedings of 2006 AOSD workshop on Open and Dynamic Aspect Languages (ODAL'O6)*, Bonn, Germany, 2006.

[52] Kim Mens and Tom Tourwé. Evolution issues in aspect-oriented programming. In Tom Mens and Serge Demeyer, editors, *Software evolution*, chapter 9, pages 203–232. Springer-Verlag, Berlin Heidelberg, 2008.

[53] Daniel L. Moise and Kenny Wong. An industrial experience in reverse engineering. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 275–284, Washington, DC, USA, 2003. IEEE Computer Society.

[54] Daniel L. Moise and Kenny Wong. Extracting facts from perl code. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 243–252, Washington, DC, USA, 2006. IEEE Computer Society.

[55] Istvan Nágy, Remco van Engelen, and Durk van der Ploeg. *Ideals: evolvability of software-intensive high-tech systems*, chapter An overview of Mirjam and WeaveC, pages 69–86. Embedded Systems Institute, TU/e Campus, Eindhoven, The Netherlands, December 2007.

[56] M.E. Nordberg III. Aspect-Oriented Dependency Inversion. In *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, OOPSLA*, Tampa Bay, FL, USA, 2001.

[57] Klaus Ostermann, Mira Mezini, and Christophe Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240, TODO, 2005. Springer.

[58] Wim De Pauw, Richard Helm, Doug Kimelman, and John M. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 326–337, New York, NY, USA, 1993. ACM Press.

[59] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[60] Tamar Richner. *Recovering Behavioral Design Views: a Query-Based Approach.* PhD thesis, University of Berne, Switzerland, 2002.

[61] Claudio Riva. Reverse architecting: An industrial experience report. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 42–50, Washington, DC, USA, 2000. IEEE Computer Society.

[62] Martin P. Robillard. Automatic generation of suggestions for program investigation. *SIGSOFT Software Engineering Notes*, 30(5):11–20, 2005.

[63] O. Rohlik, A. Pasetti, V. Cechticky, and I. Birrer. Implementing Adaptability in Embedded Software through Aspect Oriented Programming. In *IEEE Mechatronics & Robotics 2004*, pages 85–90, Aachen, Germany, September 2004.

[64] Coen De Roover, Isabel Michiels, Kim Gybels, Kris Gybels, and Theo D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 202–211, Washington, DC, USA, 2006. IEEE Computer Society.

[65] Kris De Schutter. *Aspect oriented revitalisation of legacy software through logic meta-programming.* PhD thesis, Ghent University, Ghent, Belgium, May 2006.

[66] Kris De Schutter and Bram Adams. Aspect-orientation for revitalising legacy business software. *Electr. Notes Theor. Comput. Sci.*, 166:63–80, January 2007.

[67] Harry Sneed. Encapsulating legacy software for use in client/server systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 104–119, Washington, DC, USA, 1996. IEEE Computer Society.

[68] Harry Sneed. Program comprehension for the purpose of testing. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 162–171, Washington, DC, USA, 2004. IEEE Computer Society.

[69] Harry Sneed. An incremental approach to system replacement and integration. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 196–206, Washington, DC, USA, 2005. IEEE Computer Society.

[70] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[71] Olaf Spinczyk and Daniel Lohmann. The design and implementation of aspectc++. *Know.-Based Syst.*, 20(7):636–651, 2007.

[72] Amitabh Srivastava and Alan Eustace. ATOM — A system for building customized program analysis tools. In *ACM Press SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[73] Tarja Systä. *Static and dynamic reverse engineering techniques for Java.* PhD thesis, University of Tampere, Finland, 2000.

[74] Ladan Tahvildari. *Quality-Drive Object-Oriented Re-engineering Framework.* PhD thesis, University of Waterloo, Ontario, Canada, 2003.

[75] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 107–119, Los Angeles, CA, USA, 1999. IEEE Computer Society Press.

[76] Kris De Volder and Theo D'Hondt. Aspect-Orientated logic meta programming. In *Reflection '99: Second International Conference on Meta-Level Architectures and Reflection*, pages 250–272, London, UK, 1999. Springer-Verlag.

[77] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.

[78] Andy Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis.* PhD thesis, University of Antwerp, Belgium, 2006.

[79] Andy Zaidman, Bram Adams, Kris De Schutter, Serge Demeyer, Ghislain Hoffman, and Bernard De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 91–102, Washington, DC, USA, 2006. IEEE Computer Society.

[80] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 134–142, Washington, DC, USA, 2005. IEEE Computer Society.

[81] Andy Zaidman and Serge Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–338, Washington, DC, USA, 2004. IEEE Computer Society.