Aspect-gerichte revitalisatie van legacy software adhv. logisch meta-programmeren

Aspect oriented revitalisation of legacy software through logic meta-programming

Kris De Schutter

Promotoren: Prof. dr. ir. H. Tromp
Prof. dr. T. D'Hondt

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen.

Vakgroep Informatietechnologie
Voorzitter: Prof. dr. ir. P. Lagasse
Faculteit Ingenieurswetenschappen
Academiejaar 2005–2006

UNIVERSITEIT
GENT

In memory of Ghislain.
February 14, 1943 – August 25, 2005.

The Road goes ever on and on
Down from the door where it began.
Now far ahead the Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.

J. R. R. TOLKIEN

Dedication

I feel a very unusual sensation.
If it is not indigestion,
I think it must be gratitude.

BENJAMIN DISRAELI

I would like to start of this downpour of gratitude by stating that these past few years have been a blast. I have learned more about life in general, and computer science in particular, during these last four years than in all the time before that. Though I will graciously concede that my schooling and upbringing probably had some amount of good influence as well. —It's a joke; laugh.—

If there was one reason that I have enjoyed getting my hands dirty on Cobol, it was prof. Ghislain Hoffman. As a teacher he was probably the one who got my ambitions up the most. I remember that it was thanks to a remark from him that I installed my first version of Linux, and made the switch to Windows NT. I also remember that I promptly made my website in English, if for no other reason than that he stated (in an interview for a newspaper) that our student's proficiency in languages was rapidly declining. He was right, of course: I would never have attempted to put up my site in French or German. Then there was the time that I spent four weeks discussing the architecture of a group project, before coding anything of it, much to the discomfort of my fellow students. When we related this to Ghislain during our exam he was actually quite pleased at that, which was another impulse for moving into software engineering. As my promotor I learned from Ghislain much of the realities of this discipline. —It is exactly these realities which are the driving force behind the ARRIBA project, to which I found myself assigned.— I also got a privileged view on the teaching

thereof, and found that Ghislain and I were very much on the same wavelength. It is easy to say that I, as well as my colleagues, found myself in the most ideal environment thanks to Ghislain. It is needless to say that his sudden departure from us was the single hardest blow I have had to face. And so to him I say: thank you, and I will not forget.

I am especially indebted to prof. Herman Tromp and prof. Theo D'Hondt, my current promoters, for taking over from Ghislain —not an easy task—, and making sure that this dissertation you see before you actually got finished. I thank them for their good advice, useful pointers and good humour.

Thanks also to the members of the exam-committee for agreeing to referee this dissertation: Herman Tromp, Theo DHondt, Serge Demeyer, Ralf Lämmel, Frank Gielen, Bart Dhoedt, Albert Hoogewijs and Tom Mens.

I would also like to thank Koenraad Vandenborre, who acted as a tutor on my graduation thesis, for having been foolish enough to ask me what I wanted to be doing after my graduation. I'm especially grateful to him for allowing me to make my thesis my own; a luxury which not all students had.

A big thank you goes out to everyone directly, or indirectly, involved in the ARRIBA project: prof. Serge Demeyer, Andy Zaidman, Alon Amsel, prof. Theo D'Hondt, Isabel Michiels, Wolfgang Demeuter, Dirk Deridder, as well as all the members of the user committee. A thank you also to the IWT, Flanders, for making ARRIBA possible.

Ralf Lämmel deserves a special mention as well. It were his tools (and that of his colleagues at the CWI, Amsterdam) which helped me to get a Cobol parser going; a prerequisite to all our further work. His cooperation in helping define Cobble was also invaluable. Being an ex-Cobol programmer, he had, of course, a much better understanding than me of how this language works in real life. Though what I enjoy most about working with Ralf is his sense of humour. And in that spirit I should write: "*Viva Cobol!*"

I want to thank the members (past and present) of the Software Engineering Lab —now the Ghislain Hoffman Software Engineering Lab— for their support, their help, and for generally putting up with my ranting and wild ideas: Maarten Moens, David Matthys, Stijn Van Wonterghem, Bram Adams, Jan Van Besien and Mieke Creve. I would also like to mention the members of the Formal Methods Lab, prof. Raymond Boute, Hannes Verlinde and José Sampaio Faria, with whom

we are lucky to share office space, ideas and friendship.

Some extra words of gratitude to Bram Adams, Andy Zaidman and Wolfgang Demeuter, for their useful comments on earlier versions of this text. If I managed to get it anywhere near a half-decent text, it was thanks to them.

I also enjoyed acting as tutor to the graduation thesis of several students: Ruben Miguelez Garcia, Stijn Van Wonterghem, David Tas, Jan Van Besien, Bram Adams, Stijn Baes, Bram Tassyns and Michiel Bogaert. I think it's also fair to mention those who still have to suffer me in this role: Gregory Van Vooren, Bert Van Semmertier, Maria Laura Botella Herran and Ives De Bruycker. There is something really rewarding about helping a student finish his thesis.

I wouldn't have gotten anywhere without my parents, family and friends. While my dad was always hoping for me to become a civil engineer, it was probably his own fault that I ended up in computer science: had he not bought an Apple II clone and allowed me to play with it, this dissertation might have been on the restructuring of old buildings rather than of old code. Thanks!

Finally, a great big whopping thank you to miss Kristin Waterplas, who has been putting up with my sense of humour for three years now, and still hasn't grown bored with me. Thank you for allowing me to escape myself and my work when I need to. You are the best thing that has happened to me.

Kris De Schutter
Gent, March 2, 2006

Samenvatting

Een wetenschapper is een bijzondere vogel:
eerst broedt hij en vervolgens legt hij zijn ei.

FERWERDA

DIT doctoraatswerk stelt voor om een combinatie van Aspect-georiënteerd programmeren (AOP) met Logisch Meta-programmeren (LMP) te gebruiken als hulpmiddel bij de revitalisatie van bedrijfsapplicaties. De dynamiek van dergelijke applicaties wordt gekenmerkt door een groeiende vraag naar herstructurering en integratie. Dit vergt een niet te onderschatten portie menselijk inzicht en expertise, iets wat belemmerd wordt door een gebrek aan goede documentatie van dergelijke applicaties. We stellen daarom voor om gebruik te maken van de kracht en flexibiliteit van Logisch Meta-programmeren, in combinatie met het gebruiksgemak van Aspect-georiënteerd programmeren, als hulp bij het terugwinnen van bedrijfsarchitecturen, alsook bij het omvormen en integreren van bedrijfsapplicaties.

Stelling

Bedrijfsapplicaties zijn verwezenlijkingen van welbepaalde bedrijfsprocessen, en zijn daardoor heel gevoelig voor de evolutie daarvan. Uit de toegenomen globalisatie van bedrijven, en de groeiende vraag naar interconnectiviteit tussen bedrijven, volgt eens steeds groter wordende druk tot schaalvergroting en integratie van bedrijfsapplicaties. Los van de moeilijkheden met betrekking tot de integratie van de verschillende bedrijfs*modellen* en de daaraan gekoppelde bedrijfs*processen* (wat buiten het bereik van dit doctoraatswerk valt), vormt de samenwerking tussen

bedrijfs*applicaties* zelf al een groot struikelblok: op enkele uitzonderingen na ontbreekt hiervoor de nodige documentatie en ondersteuning.

In het algemeen geldt dat de data-opslagplaatsen en de lopende applicaties de enige echte beschrijving vormen van de informatiestromen en structuren. Dit betekent dat alleen de daadwerkelijke gegevens (en de manier waarop ze verwerkt worden) samen met de broncode van de applicaties de enige betrouwbare bron van informatie vormt.

Het samensmelten van bedrijfsapplicaties zal altijd menselijke expertise vereisen. Spijtig genoeg, in omgevingen waar zo los omgesprongen wordt met middelen, kan dergelijke expertise nooit ten volle benut worden.

Als lid van het ARRIBA[1] project, kijken we hoe Aspect-georiënteerde software-ontwikkeling ons kan helpen bij deze problematiek. Meer specifiek stellen we dat de combinatie van Logisch Meta-programmeren en Aspect-georiënteerd programmeren helpt bij het terugwinnen van bedrijfsarchitecturen, alsook bij het hervormen en integreren van bedrijfsapplicaties.

AOP voor Cobol

Wanneer we praten over legacy omgevingen, bevinden we ons meestal in de context van Cobol. Dit wordt ondersteund door bevindingen binnen het ARRIBA project [MDTZ03], alsook door een aantal statistieken die de Gartner groep naar voor heeft gebracht: 75% van bedrijfsdata wordt verwerkt door Cobol, met 180–200 miljard aan lijnen broncode wereldwijd te vinden, en 15% van alle nieuwe applicaties wordt geschreven in Cobol. Het is duidelijk dat Cobol de grootste speler is.

Cobble [LS05a] is onze Aspect-georiënteerde uitbreiding voor Cobol. Het omvat LMP als een "pointcut"[2] taal, en deze wordt gelinkt aan AOP met behulp van een mechanisme van *bindingen*. Het volgende voorbeeld van een tracerings-aspect verduidelijkt dit:

```
1 MY-PROCEDURE-TRACING SECTION.
    USE AROUND PROCEDURE
3   AND BIND VAR-NAME TO NAME
    AND BIND VAR-LOC TO LOCATION.
```

[1] *Architectural Resources for the Restructuring and Integration of Business Applications*; een GBOU project van het IWT, Vlaanderen. (`http://arriba.vub.ac.be/`)

[2] Beschrijving van een interessant punt binnen de uitvoering van een applicatie.

```
5 MY-TRACING-ADVICE.
   DISPLAY "Before␣procedure␣", VAR-NAME,
7          "␣at␣", VAR-LOC.
   PROCEED.
9  DISPLAY "After␣procedure", VAR-NAME,
           "␣at␣", VAR-LOC.
```

Deze notatie blijft dicht bij de stijl van Cobol, en dit om een betere integratie met de taal te verwezenlijken. We zien dus dat advies niets meer is dan procedures in standaard Cobol. Het advies zelf leeft in een sectie (lijn 1), met de quantificatie van het advies uitgewerkt in een `USE` opdracht. Deze laatste steunt op de reeds beschikbare, zij het beperkte, syntax voor het uitdrukken van "crosscutting concerns"[3]. We zien dat bovenstaand advies wordt uitgevoerd "rond" alle procedures (lijn 2). We extraheren van deze procedures de naam en locatie (lijnen 3 en 4), die dan beschikbaar zijn voor gebruik binnen het advies (lijn 6–10). We voorzien daarbij een `PROCEED` opdracht voor het oproepen van het originele "join point"[4].

AOP voor ANSI-C

Naast een mastodont als Cobol blijft er voor de andere talen niet veel marktaandeel meer over. Niettemin blijkt C een belangrijke en veelgebruikte taal —o.a. binnen de gebruikerscommissie van het ARRIBA project.

Vanuit dit oogpunt startte ons werk aan *Wicca* [Won04], een Aspect-georiënteerde uitbreiding voor C. Met het proceduraal programmeren van C enerzijds, en het "zin"-georiënteerd[5] programmeren van Cobol anderzijds, bestrijken we zo toch een uitgebreid spectrum aan legacy talen.

Volgende code is een voorbeeld van een tracerings-aspect in Wicca:

```
int advice on (.*) && ! on (printf) {
2  int r = 0;
   printf ("before␣%s\n", this_joinpoint()->name);
4  r = proceed ();
   printf ("after␣%s\n", this_joinpoint()->name);
```

[3] Overlappende facetten van een applicatie.

[4] Een uitvoeringspunt binnen de applicatie.

[5] "Zin" is hier te lezen als een zin in een tekst, als een grammaticaal gegeven.

```
6   return r;
  }
```

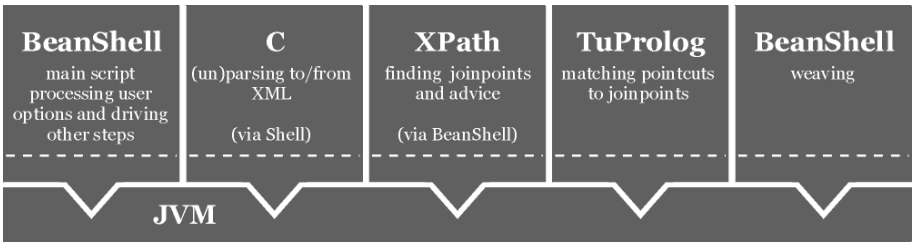
Wicca blijft dicht bij de syntax en semantiek van AspectJ. Het bovenstaande advies werkt "rond" alle procedures, behalve die met als naam "printf" (lijn 1). Adviezen bestaan uit gewone C code, en krijgen toegang tot de context van het join point met behulp van de daarvoor bedoelde `this_joinpoint()` procedure (lijnen 3 en 5). Activatie van het originele gebeurt met een oproep aan `proceed` (lijn 4).

Als Wicca eenvoudiger en beperkter lijkt dan Cobble, dan is dit correct. Wicca is dan ook ontwikkeld vóór Cobble. Het maakt daardoor nog geen gebruik van LMP, wat ons ervan weerhoudt om dit advies op een generieke wijze te schrijven. Het bovenstaande advies werkt namelijk uitsluitend als alle geselecteerde join points een `int` waarde teruggeven. Dit is naturlijk een onrealistische voorwaarde voor de gemiddelde applicatie. Dit toont dan ook al meteen één gebied waar LMP nuttig kan zijn.

Hier volgt het zelfde voorbeeld uitgewerkt in *Aspicere* [AT05]. Dit is een opvolger van Wicca, onder ontwikkeling door Bram Adams, die wel LMP ingebed heeft.

```
1 Type around tracing (Type) on (Jp):
    ! call (Jp, "printf")
3   && type (Jp, Type)
    && ! str_matches ("void", Type)
5 {
    Type r = 0;
7   printf ("before␣%s\n", Jp->name);
    r = proceed ();
9   printf ("after␣%s\n", Jp->name);
    return r;
11 }
```

Net als in Cobble slagen we er nu in om informatie te verkrijgen met behulp van bindingen. `Type` is zo'n binding, die we hier gebruiken om ons advies toepasbaar te maken op alle mogelijke procedures, ongeacht het type van de teruggegeven waarde (afgezien van `void`, dat buiten de normale typering valt).



Figur A. Raamwerk zoals ingesteld voor Cobble.

Transformatie-raamwerk

Voor de weving van applicaties hebben we een raamwerk voorzien dat gebaseerd is op XML-voorstellingen van broncode. Dankzij het XML-formaat, en met wat hulp van de JVM, was het mogelijk om een mix van technologieën te integreren tot een samenhangend en functioneel geheel. (Figuur A toont hiervan een voorbeeld binnen de context van Cobble.) Het vinden van adequate parsers voor de verschillende legacy talen vormde hierbij nog de zwaarste opdracht, zeker in het geval van Cobol.

De omvorming van de XML-voorstellingen heeft één beperking: het eindresultaat moet een geldig programma voorstellen. Dit vormde een struikelblok bij Cobol, waar deze voorwaarde ons belette om individuele lees- en schrijfopdrachten te selecteren. Data items in Cobol kunnen niet vervangen worden. In C kunnen we die tenminste nog omzetten naar de aanroep van een functie. Deze beperking zouden we kunnen omzeilen door te opteren voor het weven van machinecode, maar daarmee zouden we tevens onze platformonafhankelijkheid verliezen.

Het waarom van LMP

Om het schrijven van generiek advies mogelijk te maken hebben we nood aan reflectie op het basis-programma. Bij talen als Java, die voorzien in een API voor reflectie, moet de AOP taal daarbij niets speciaals meer doen —hoewel bepaalde onderzoeken dit in vraag stellen [GB03, KR05, De 01, HC03, HU03].

Cobol en C beschikken echter niet over dergelijke mogelijkheden, wat ons er inderdaad van weerhoudt om generiek advies te schrijven.

```
{ LINKAGE SECTION.
  01 METHOD-NAME PIC X(30) VALUE SPACES. },

findall(
  [Name, Para, Wss],
  ( paragraph(Name, Para),
    slice(Para, Slice),
    wss(Slice, Wss)
  ),
  AllInOut
),

max_size(AllInOut, VirtualStorageSize),
{ 01 VSPACE PIC X(<VirtualStorageSize>). },

all(member([Name, Para, Wss], AllInOut), (
  {  01 SLICED-<Name> REDEFINES VSPACE.},
  all( (record(R, Wss), name(R, RName)), (
    clone_and_shift(R, "<RName>-<Name>", SR),
    { <SR> }
  ))
))
```

Figuur B. Encapsulatie van procedures (uittreksel).

Dit ondervonden Bruntink, Van Deursen en Tourwé in hun werk voor ASML [BvDT04], de marktleider op het gebied van systemen voor lithografie. De aspecttalen moeten dit gebrek dus goed maken. LMP, samen met een eenvoudig template mechanisme, maakte dit voor ons mogelijk.

Bovendien biedt LMP ons de mogelijkheid om bedrijfskennis te registreren in een herbruikbare vorm. Zonder deze mogelijkheid zien ontwikkelaars zich gedwongen om deze kennis met de hand toe te passen binnen applicaties. LMP kan (heel eenvoudig) aangewend worden om dit probleem op te lossen. In samenspel met AOP kunnen we dit op een directe manier benutten.

Bij het inkapselen van bedrijfslogica bereikten we echter wel de grenzen van wat LMP kan doen in AOP. We waren niet in staat om de complexe en crosscutting gegevensdefinities uit te drukken, die nodig

waren om het probleem op te lossen. Dankzij LMP in AOP konden we verregaand werken rond gedrag, maar niet rond structuur.

De oplossing kwam er door het omdraaien van de AOP/LMP vergelijking. Door AOP in LMP in te passen werd het neerschrijven van dergelijke structuren mogelijk. Deze techniek staat heel dicht bij het werk rond parametrische introducties, zoals in [HU03] en [BMD02]. Figuur B. toont een deel van onze oplossing voor het inkapselen van bedrijfslogica, als voorbeeld.

Een extra beperking kwam voort uit de beperkte ondersteuning van Cobol voor data definities. Ondanks de mogelijkheid tot heel complexe en specifieke definities van dataformaat —probeer maar eens een getal met exact 11 decimalen te definiëren in Java—, laat het niet toe deze definities te hergebruiken. Het is bijvoorbeeld niet mogelijk om te weten te komen welke data items datums voorstellen. Elk getal van zes of acht cijfers kan een datum zijn, maar is het niet noodzakelijk. Het is al evenmin zo dat getallen van andere lengte daarom geen datum kunnen voorstellen.

Dit weerhoudt ons er nogmaals van om generiek advies te schrijven. Hoe quantificeren we over types als er geen zijn? Het is met LMP dan wel mogelijk om alle data items aan te duiden die datums bevatten, maar erg handig is dat niet. We vinden dan ook dat de kracht van AOP en LMP soms wordt ingeperkt door de onderliggende legacy taal.

Validatie

We bekeken vier problemen rond legacy software van dichtbij. Eerst toonden we aan hoe we, met behulp van een eenvoudig traceer aspect, dynamische analyses van legacy systemen mogelijk maakten. We deden dit binnen een reële case studie. [ZAD+06, ZAD05, ADZ05] LMP werd daarbij gebruikt om het traceer advies zo generisch mogelijk te maken.

Als tweede werkten we een scenario uit dat het terugwinnen van bedrijfslogica als doel had. [MDDH04] AOP en LMP werden daarbij toegepast op verschillende manieren: van slim en geconcentreerd traceren, tot verificatie van veronderstellingen en, uiteindelijk, het terugwinnen van de logica.

Het derde probleem draaide rond de inkapseling van legacy applicaties, op basis van een generisch aspect. Een beperkte vorm van

inkapseling kon heel eenvoudig uitgewerkt worden, maar de doorgedreven vereisten, zoals besproken in [Sne96a], dwongen ons er toe om een andere aanpak te bekijken. Dankzij het plaatsen van AOP binnen LMP konden we ook dit opgelost te krijgen, en dit zonder echte problemen.

Bij het vierde probleem, dat van Y2K, kon onze bron-naar-bron weving niet voorbij aan de beperkingen van de onderliggende taal. De semantiek van Cobol, met name de beperkingen bij typering, vormde een te groot struikelblok. Binnen C daarentegen bleek het Y2K38 probleem beter handelbaar, juist omdat typering daar aanwezig is.

Conclusie

Door het inpassen van AOP en LMP binnen legacy talen, en daardoor ook binnen bestaande bedrijfsomgevingen, beschikken we over een flexibele en krachtige vorm van gereedschap. Er is daarbij geen nood om af te stappen van bestaande ontwikkeltechnieken; onze oplossing kan gebruikt worden in samenspel ermee.

Daarenboven kunnen we dankzij LMP bedrijfsconcepten en architecturale beschrijvingen van bedrijfsapplicaties uitdrukken op een exploiteerbare manier. Deze werkwijze maakt het mogelijk om met applicaties te werken op een hoger niveau van abstractie, en zal hopelijk leiden tot een betere definitie van de architectuur.

Summary

The covers of this book are too far apart.

AMBROSE BIERCE

THIS work proposes the combination of Logic Meta-programming and Aspect-oriented Programming as a tool to tackle the ills of legacy business applications. The dynamics for such applications are characterized by a need for restructuring and integration at a much larger scale than was previously the case. This requires a non-trivial amount of human insight and experience, something which is hampered by a general lack of good documentation of these applications. We therefore propose to adopt the power and flexibility of Logic Meta-programming, combined with the ease-of-use of Aspect-oriented Programming, to aid in the recovery of business architectures, as well as in the restructuring and integration of business applications.

Thesis

Business applications are instantiations of specific business processes, and as such they are highly susceptible to the evolution thereof. With increased globalisation of enterprises, and ever greater demand for interconnectivity between companies, comes increasing pressure to scale up and integrate business applications. Aside from difficulties in integrating the different business *models* and their associated business *processes* (which is well outside the scope of this dissertation), getting the business *applications* to cooperate is a major hurdle: in all but a few cases the documentation and support of these applications is insufficient (or even absent).

In general, the data repositories and the running programs provide the only true description of the information structures and applications they implement. Hence, the actual data (and the way it is handled) and the source code to those applications form the only dependable documentation.

Merging business applications will always require the application of human expertise. Unfortunately, in an environment where its assets are so poorly understood this expertise can never be fully exploited.

As part of the ARRIBA[6] project, we focus on how the emerging paradigm of Aspect-oriented Software Development can be applied to this problem. More precisely, we claim that the combination of Logic Meta-programming and Aspect-oriented Programming aids in the recovery of business architectures, as well as in the restructuring and integration of business applications.

AOP for Cobol

When talking about legacy environments, we mostly find ourselves in the realm of Cobol. This is corroborated by our findings within the ARRIBA research project [MDTZ03], as well as by the numbers put forward by the Gartner group: 75% of business data is processed in COBOL, with 180–200 billion LOC in use worldwide, and 15% of new applications written in COBOL. It is clear that Cobol is *the* major player.

Cobble [LS05a] is our aspectual extension to Cobol, which embeds LMP as a pointcut language, and binds it to AOP through a mechanism of bindings. Consider the following tracing aspect as an example:

```
MY-PROCEDURE-TRACING SECTION.
  USE AROUND PROCEDURE
  AND BIND VAR-NAME TO NAME
  AND BIND VAR-LOC TO LOCATION.
MY-TRACING-ADVICE.
  DISPLAY "Before␣procedure␣", VAR-NAME,
          "␣at␣", VAR-LOC.
  PROCEED.
  DISPLAY "After␣procedure", VAR-NAME,
          "␣at␣", VAR-LOC.
```

[6] *Architectural Resources for the Restructuring and Integration of Business Applications*; a GBOU project sponsored by the IWT, Flanders. (`http://arriba.vub.ac.be/`)

The syntax remains close to that of Cobol, in order to allow better integration. I.e., advice bodies are, in effect, regular Cobol procedures. The advice itself is a "section" (line 1), with the quantification encoded in a `USE` statement. This latter leverages the existing, though very limited, syntax for crosscutting concerns. As can be seen we advice around all procedures (line 2). We then extract the name and location of that procedure (lines 3 and 4), which are available for use within the advice body (lines 6–10). We provide a `PROCEED` statement (line 8) for invoking the original join point.

AOP for ANSI-C

With only a limited market share left by Cobol for its competitors, C does stand out as an important legacy language. Again, when turning to the ARRIBA user committee, we find C is in active use, more so than other languages.

We therefore worked on *Wicca* [Won04], an aspectual extension for C. Between the procedural paradigm of ANSI-C, and the statement-oriented ("English") paradigm of Cobol, we feel we have thus covered quite a broad spectrum of legacy languages.

Here is a very basic tracing aspect written in Wicca:

```
int advice on (.*) && ! on (printf) {
  int r = 0;
  printf ("before␣%s\n", this_joinpoint()->name);
  r = proceed ();
  printf ("after␣%s\n", this_joinpoint()->name);
  return r;
}
```

Wicca remains close to AspectJ's [KHH+01] syntax and semantics. As such, the above aspect might look familiar. It advices around any procedure, except those named "printf" (line 1). The body itself is pure C code, which is able to access join point context by means of a call to the `this_joinpoint()` procedure (lines 3 and 5). Activation of the original join point occurs through a `proceed` call (line 4).

If Wicca appears simpler and more limited than Cobble, there is a reason for this: it was developed some time *before* Cobble. Hence, it is not using LMP, nor can we use it to write generic advice. I.e., the above example will only work when procedures return an `int` value. This is,

of course, not very useful for tracing. It does however show one area where LMP can help.

Here is the same example, but now written down in *Aspicere* [AT05], a follow-up of Wicca, being developed by Bram Adams, which does support LMP:

```
1 Type around tracing (Type) on (Jp):
    ! call (Jp, "printf")
3   && type (Jp, Type)
    && ! str_matches ("void", Type)
5 {
    Type r = 0;
7   printf ("before␣%s\n", Jp->name);
    r = proceed ();
9   printf ("after␣%s\n", Jp->name);
    return r;
11 }
```

Much as in Cobble we are now able to extract information through bindings. I.e. `Type` is a binding which is used to make our advice type independent (though we do still need an advice for `void` procedures).

Transformation framework

On the weaving side, we have built a transformation framework based on XML representations of source code. Through the XML format, and with a little help from the JVM, we were able to bring together a mix of disparate technologies and integrate them into a cohesive and functional whole. (Figure A. shows an example of this in the context of Cobble.) The hardest part lay in finding decent parsers for the legacy languages, which proved especially tricky in the case of Cobol.

Transforming the XML representations is only limited in that the end-result of that transformation must be a valid program. This proved to be an obstacle for Cobol, where this well-formedness keeps us from weaving individual getters and setters. Indeed, data items can not be replaced by anything other than data items. In C we can at least replace any reference to a variable with a call to a procedure. Not so in Cobol. Machinecode weaving can hold the answer to this limitation for Cobol, but it would mean sacrificing our platform-independence.

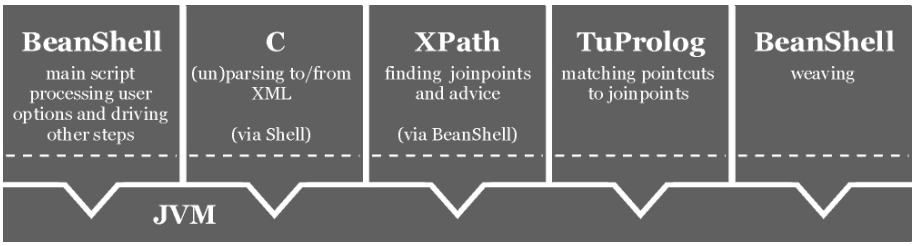


Figure A. Framework setup for Cobble.

The why of LMP

In order to write generic aspects we need to be able to reflect on the base programs, to extract information about that base program. With languages such as Java, which offer a reflection API natively, the aspect language need not do anything special —though there is research that questions this [GB03, KR05, De 01, HC03, HU03].

Cobol and C have no reflective capabilities, which indeed limits us from writing generic advice. This was experienced by Bruntink, Van Deursen and Tourwé in their work for ASML, the world market leader in lithography systems. [BvDT04] The associated aspect languages must therefore make up for this. We have shown that LMP, together with a simple template mechanism, can do this for us.

Furthermore, the ability of LMP to record business knowledge can be of great advantage to legacy software. Where languages lack reflective capabilities, they must also lack the ability to encode information about programs. Hence, programmers are forced into hacks and workarounds. LMP can be used —very simply— to encode a wide spectrum of information. In cooperation with AOP we were then able to exploit this in a direct way.

When tackling the encapsulation of business logic, however, we encountered the limits of expressiveness when embedding LMP in AOP. We were not able to describe the complex, though crosscutting, data definitions required to solve the task at hand. When embedding LMP in AOP, we can do great things when it comes to behaviour, but not when it comes to structure.

We therefore turned the AOP/LMP equation around. By embedding AOP in LMP, we can write down the complex crosscutting structures, a technique which is close to the idea of parametric introductions,

```
  { LINKAGE SECTION.
24   01 METHOD-NAME PIC X(30) VALUE SPACES. },

26 findall(
     [Name, Para, Wss],
28   ( paragraph(Name, Para),
       slice(Para, Slice),
30     wss(Slice, Wss)
     ),
32   AllInOut
  ),
34
  max_size(AllInOut, VirtualStorageSize),
36 { 01 VSPACE PIC X(<VirtualStorageSize>). },

38 all(member([Name, Para, Wss], AllInOut), (
     {  01 SLICED-<Name> REDEFINES VSPACE.},
40   all( (record(R, Wss), name(R, RName)), (
       clone_and_shift(R, "<RName>-<Name>", SR),
42     { <SR> }
     ))
44 ))
```

Figure B. Full procedure encapsulation (excerpt).

as in [HU03] and [BMD02]. Figure B. shows an excerpt of this solution as applied to encapsulation of business logic.

Another limitation for generic advice lies in Cobol's support for data definitions: it allows for very detailed descriptions of anything from numbers to strings —consider having to write a number in Java which has exactly 11 decimals—, yet it does not allow the reuse of these definitions. There is, for instance, no easy way to figure out which data items are dates. Any item holding six digits may be a date or it may not. Nor is it said that any item which is not made up of six or eight digits must therefore not be a date…

This again poses a limitation for writing generic advice: how do we quantify over types (e.g. dates) when there are none? The only solution seems to be that we should write down, as business knowledge, which data items pertain to a certain type. This is tedious at best. So we

find that despite the strength of AOP and LMP, the legacy languages sometimes get the overhand.

Validation

We have taken a closer look at four problems with legacy software. First, we have shown how, through a simple tracing aspect, we managed to enable dynamic analyses in a real case study. [ZAD+06, ZAD05, ADZ05] LMP was used to make the tracing advice as generic as possible by overcoming a lack of reflection in the base language.

Second, we worked through a scenario for the recovery of business knowledge from a legacy application. [MDDH04] AOP and LMP were applied here in several ways: from smart and focused tracing, to verification of assumptions and, ultimately, the rediscovery of logic.

Thirdly, we tackled the problem of encapsulating legacy applications using a generic aspect. A basic form of encapsulation could be done easily, but the full-on approach, as described in [Sne96a], required us to take a step back. Still, by placing AOP within LMP, we were able to tackle even this problem, and relatively straightforward at that.

Only in the fourth problem, Y2K, our source-to-source weaving approach was held back by the limitations of the base language. As it is, the semantics of Cobol, especially its lack of typing, present too much of a roadblock. In C, the Y2K38 problem can still be managed reasonably, precisely because it features such typing.

Conclusion

By having embedded AOP and LMP in legacy languages, and hence in existing business environments, we now have at our disposal a flexible toolchain. There is no requirement to move away from the existing development techniques; the toolchain can be used in addition to them.

Furthermore, through LMP we are able to express business concepts and architectural descriptions of business applications. This makes it possible to work with applications at a higher level of abstraction, which we hope will also encourage better architectural descriptions to emerge.

Contents

List of Figures

Chapter 1

Introduction

It's a dangerous business, Frodo, going out your front door.

J. R. R. TOLKIEN

IN this dissertation, we propose the combination of Logic Meta-programming and Aspect-oriented Programming as a tool to tackle the ills of legacy business applications. The dynamics for such applications are characterized by a need for restructuring and integration at a much larger scale than was previously the case. This requires a non-trivial amount of human insight and experience, something which is hampered by a general lack of good documentation of these applications. We therefore propose to adopt the power and flexibility of Logic Meta-programming, combined with the simplicity of Aspect-oriented Programming, to aid in the recovery of business architectures, as well as in the restructuring and integration of business applications.

1.1 Problem statement

This section sets the stage in which the contributions of this dissertation were elaborated. The context, put briefly, is that of business applications which, being instantiations of specific business processes, are highly susceptible to the evolution of these processes. Aside from difficulties in the evolution of different business models and their associated processes, getting these applications to evolve turns out to be far from trivial.

1.1.1 Context

The research which is being expanded upon in this dissertation is part of an IWT-Flanders research project, named ARRIBA. Short for *Architectural Resources for the Restructuring and Integration of Business Applications*, its aim is:

> "*to provide a methodology and its associated tools in order to support the integration of disparate business applications that have not necessarily been designed to coexist.*"

While this would align the ARRIBA project closely with *Enterprise Application Integration* (EAI) techniques, the project proposal further states that:

> "*the current EAI-inspired methods and tools fail to provide a satisfactory setting in which to proceed.*"

So there is an incentive to look for solutions outside of the existing EAI-toolchain. To this end the project is supported by a consortium of research groups from the Vrije Universiteit Brussel, the University of Antwerp and the University of Ghent, together with a user committee consisting of eight companies:

- Inno.com: an ICT expertise center, advising and assisting its clients and partners to cope with their technology and business issues. (`www.inno.com`)
- Anubex: an expert in application modernisation through software conversion and application migration. (`www.anubex.com`)
- Banksys: manages the Belgian network for electronic payments for banks, merchants and consumers. (`www.banksys.be`)
- Christelijke Mutualiteit: largest Belgian social security provider. (`www.cm.be`)
- KAVA: a non-profit organization grouping over a thousand Flemish pharmacists. (`www.kava.be`)
- KBC Bank & Verzekering: banking and insurance company. (`www.kbc.be`)

- PEFA.com: services European fresh fish companies by enabling fish auctions to work over the internet. (`www.pefa.com`)
- Toyota Motor Europe: European branch of the Toyota motoring company. (`www.toyota.be`)

At the heart of this project lies the recognition that modern business applications are characterized by a need for restructuring and integration, and that at a much larger scale than ever before. [MDTZ03, Tic01] This follows from the restructuring and integration of organizations themselves, as they, among other things, strive to merge their activities and, hence, their ICT infrastructure.

1.1.2 Change happens

The recognition that business applications are faced with a need for restructuring and integration is not new. There exists a widely accepted law stated that systems will either have to evolve (change) or be scrapped:

> "*An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.*" [Leh96]

Lehman first postulated this, what has become known as the "first law of software evolution", in the mid seventies. [Leh74] E-type systems he considers to be, broadly speaking,

> "*software systems that solve a problem or implement a computer application in the real world.*" [Leh96]

The same principle is also found outside computer science. Evolution theory talks about the *Red Queen Equilibrium*. This is due to a quote from Lewis Carroll's famous book '*Alice through the looking glass*'. In it Carroll has the Red Queen remark: "*here, you see, it takes all the running you can do, to keep in the same place.*" The principle reflects that in order to keep up with an evolving environment you have to keep investing energy to maintain your place within it.

Examples of the driving forces behind this process of change —in the relatively safer world of software systems, that is— are:

- redefinition of corporate strategies,

- take-overs and mergers,
- moving from data processing to service models (brought on by the ever growing internet), [GKMM04]
- hardware (eg. mainframes) which will no longer be supported by manufacturers,
- changes in legislation which force an update of business rules and, possibly even, the workflow.

To this we can add such unforeseen global forces as the adoption of the Euro, or even the Y2K problem. [Leh98]

Reaction time to these changes in business environments is critical. New laws and regulations must be met by given deadlines, competitor advances must be matched immediately, and customer wishes must be fulfilled within a reasonable timeframe. [Sne96b]

This continuous process of modification results in a situation where several software systems will have to be changed or collaborate in ways that were never (or could never have been) anticipated in their original design.

1.1.3 Coping with change

So, now knowing that a program must be evolved lest it become more and more useless, we are faced with a second problem:

> "*As a program is evolved its complexity increases unless work is done to maintain or reduce it.*" [Leh96]

Apart from a "status-quo" scenario, in which the business adapts to the software, a number of other approaches to cope with this problem are frequently seen:

1. Rewrite the application from scratch using a new set of requirements. [Ben95]
2. Reverse engineer the application first, and only then rewrite it from scratch. [Ben95]
3. Refactor the application. One can refactor the old application, without migrating it, so that change requests can be efficiently implemented; or refactor it to migrate it to a different platform.

4. Wrap it. Turn the old application into a component in, or a service for, a new software system. In this scenario, the software still delivers its useful functionality, with the flexibility of a new environment. [Ben95, Sne96a]

5. A mix of the previous options, in which the old application is seriously changed before being set-up as a component or service in the new environment.

Certainly for all scenarios but the first, the software engineer would ideally like to have:

- a good understanding of the application in order to start his or her reengineering operation (or in order to write additional tests before commencing reengineering), [Sne04]

- a well-covering (set of) regression test(s) to check whether the adaptations that are made are behaviour-preserving. [DDN03]

Either way, any sort of change will require a great deal of human insight and experience. [Leh98]

1.1.4 Hurdles and inhibitions in evolving software

Having learned that software must evolve, and that during this evolution we must take care that its complexity remains manageable, we must now consider how to go about this. When doing so, we find that companies are faced with some recurring hurdles and inhibitions. Within the ARRIBA project, we found this to include the following things. [MDTZ03]

Proven technology. Large organisations, especially those which rely on continuous uptime and timely service for their survival, depend heavily on proven technology. This is why mainframe-technology remains of such importance. While UNIX-like systems are also in use, they are considered less reliable, and are therefore avoided for supporting essential business operations.

User resistance. There may be user resistance to change as well. Existing software may be very reliable and responsive to customer

needs (and those customers may be reliant on undocumented features). [Ben95] In the short term, a replacement system may be less reliable and require its customers to do a lot of relearning and rework. [Sne05]

Old programming languages. When we turn to the ARRIBA user committee, mentioned in section 1.1.1, we find that most of their assets are written in (some dialect of) Cobol. [MDTZ03] This finding is corroborated by the numbers[1] presented to us by the Gartner group:

- 75% of business data is processed in COBOL.
- There are 180 billion to 200 billion lines of COBOL in use worldwide.
- 15% of new applications are written in COBOL.

These numbers are in stark contrast with the roll-out of new developers which are familiar with Cobol.

Human resources. Where technology evolves quickly and has a short life-span, humans have to keep up. For large companies with hundreds of developers this means a huge investment in human resources. This is, of course, an important issue when adopting technology: while new developments may be of interest, one also has to be able to support them. That means that you either need to retrain your staff, or replace them. Under such circumstances, sticking close to what you are good at is a valuable strategy.

Data storage. Large-scale software systems need to deal with large-scale data. Unfortunately, in many such systems, the use of a Relational Database Management System (RDBMS) is still rare. Proprietary, flat-file systems are more likely to be the norm, even if use of an RDBMS has received priority.

Data models. Aside from storing data there is also the problem of interpreting it. In large companies there usually is no central ownership of data or information. This means that data information models can

[1] These numbers are for the year 2003.

(and will) develop in different ways. When the time comes to realign them, subtle differences in interpretation of the data can bring on big problems.

Business-driven. IT development is usually well-defined within companies, and a lot of attention is paid to it. It is not, however, technology-driven. IT developments which do not have a relevant business case will simply not be realized. This means that there is an inherent mismatch in how technology evolves versus how businesses evolve.

Again, it should be clear that there is an important need for human insight and experience into the existing business applications, if we are to work with, or overcome, the cited problems.

1.1.5 Pitfall: lacking documentation

It now is obvious that, in order to evolve software in a satisfactory way, we are in need of a thorough understanding of the applications supporting the business processes. Yet, in practice, we find this understanding to be lacking. Let's consider why that might be so:

- Business applications are no longer understood, as major mission-critical software was developed —sometimes decades ago— by programmers who have moved on to other projects, or that are no longer working at the company. The maintainers were not the designers. [CC90]
- Code is badly structured and poorly documented. The amount of code is huge [Ben95] and has been adapted many times for several reasons (switching platforms, year 2000 conversions, transition to the Euro currency, etc.).
- Evolution of an implementation causes it to drift away from the original architecture. [Men00, PW92, dOC98] Keeping the documentation synchronized with those evolutionary changes did not always happen. [CC90, MW03]
- Logic (whether application logic or business rules) is spread out over the entire application. Legacy languages often support only limited modularity mechanisms. Therefore complex logic has to

be manually distributed over the programs; a process which is prone to errors. [DD99b]

- Legacy programming languages are no longer (well) understood. COBOL is not popular with the new generation of programmers, nor is it being actively taught to students. Dijkstra's famous quote, "*the use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence*", captures the problem nicely.

- Use of different (versions of) features offered by different vendors makes it harder to understand code, and harder to evolve that code. It also makes it harder for toolbuilders to support these different platforms.

In general, the data repositories and the running programs themselves are the only true description of the information structures and applications they implement. Hence, the only dependable documentation is the actual data (and the way in which it is handled) and the source code itself (barring extreme cases where code is out-of-sync, or even gone). [Ben95]

1.1.6 Legacy systems

We have seen sofar that, on the one hand, software needs to evolve, and that this requires knowledge of the applications involved, but that, on the other hand, this knowledge is lacking. This problem is often referred to as the problem of "*legacy systems*". The term, legacy, brings with it an intuitive understanding of the problem, yet it is hard to find a single, fitting definition. Bennet defines legacy systems informally as,

> "*large software systems that we don't know how to cope with but that are vital to our organisation.*" [Ben95]

Brodie and Stronbraker describe it as,

> "*any information system that significantly resists modification and evolution to meet new and constantly changing business requirements.*" [BS95]

Nicholas Gold puts it as follows,

> "*Legacy software is critical software that cannot be modified efficiently. A legacy system is a socio-technical system containing legacy software.*" [Gol98]

Put more succinct, we could say that: if software moves you forward it is an asset; if it holds you back it is legacy. But rather than crank out another definition for benefit of this dissertation, we will focus on two properties that the above definitions share:

1. The software (or system) is in a state where any modification requires a large (disproportionate) amount of effort.

2. The software (or system) is of such value to its stakeholders that they can not put it aside.

1.1.7 Goal

As we have seen, companies are faced with software that needs to be evolved for various reasons, yet for which it is hard (disproportionately so) to do this. Putting this software aside is not a reasonable option to its stakeholders. We therefore have to make modification of existing applications easier. This includes support for helping with the modification itself, but also to make up for missing knowledge about the system, which is crucial for proper evolution. Ideally, we are looking for a technique which can cover both of these concerns. We state that Aspect-oriented Programming, combined with Logic Meta-programming is such a technique. The following section will delve into the reasoning behind this statement.

1.2 Solution space

This section presents the techniques of Declarative Meta-programming, Logic Meta-programming and Aspect-oriented Programming. It expands on the reasoning for applying these techniques to the problems of evolving (legacy) business software (discussed in the previous section), and as such provides the background for the upcoming chapters on aspect-oriented extensions of ANSI-C and Cobol.

1.2.1 Declarative meta-programming

Hill and Lloyd summarise *declarative* programming as being

> "*much more concerned with writing down* what *should be computed and much less concerned with* how *it should be computed.*" [HL94]

The idea that the computation which has to take place is deduced from declarations lies at the heart of declarative programming. The actual way in which the computation proceeds should be of no concern to the programmer. This is believed to make declarative programs easier to understand and reason about than their imperative counterparts.

A *meta*-program, on the other hand, is a program that takes one or more other programs as data. [Bow98] Or, to paraphrase Maes [Mae87]: "All programs are about something. A meta-program is about programs." A compiler is an excellent example of a meta-program.

Declarative Meta-programming (DMP) then combines the best of both worlds: it is a program that acts upon other programs, but it does so by declaring what should happen to them, rather than how it should be done. Or, put differently, DMP is using a declarative programming language at the meta-level to reason about and manipulate programs built in some underlying base language.

DMP requires that the reification between the DMP-language and the base language is made explicit, allowing base-level programs to be expressed as declarations at the meta-level. [MMW00]

1.2.2 Logic meta-programming

Logic Meta-programming (LMP) is an instantiation of Declarative Meta-programming where the meta-language is based on a logic programming language. These are based on first-order logic with restrictions that make efficient proof procedures possible. Logic programming developed out of work on automated theorem proving in the 1960s, particularly through the work of Robinson who introduced the resolution principle. [Rob65]

In their most basic form, logic programs are finite sets of Horn clauses, which are formulas of the form $A \leftarrow W$, where W is a conjunction of atoms. The following presents a typical example of this in Prolog:

```
parent(A,B) :- father(A,B).
parent(A,B) :- mother(A,B).

ancestor(A,B) :- parent(A,B).
ancestor(A,B) :- parent(A,X),
                 ancestor(X,B).
```

These predicates define when one item is an ancestor of another. They can be read as:

- A is a parent of B when A is a father of B. (Line 1.)
- A is a parent of B when A is a mother of B. (Line 2.)
- A is an ancestor of B when A is a parent of B. (Line 4.)
- A is an ancestor of B when there exists some X for which A is a parent and which itself is an ancestor of B. (Lines 5 and 6.)

Now, given the facts:

```
father(marcel,an).
mother(an,antje).
```

A deduction that `ancestor(marcel,antje)` holds, can be made:

```
ancestor(marcel,antje)
  :- parent(marcel, X), ancestor(X, antje)
  :- parent(marcel, an), ancestor(an, antje)
  :- father(marcel, an), parent(an, antje)
  :- father(marcel, an), mother(an, antje).
```

How that deduction actually proceeds is, in principle, of no importance. While logic programming languages do, of course, have an operational semantics, all that matters is that the deduction can be made.

Logic programming becomes Logic Meta-programming when the facts about which we define predicates are themselves structures of some program. For instance, by using facts on which classes extend which other classes, we could write:

```
inherits(Sub,Base) :- extends(Sub,Base).
inherits(Sub,Base) :- extends(Sub,X),
                      inherits(X,Base).
```

From this we could then, for example, start to reason about inheritance chains. This is, of course, very similar to the parent/ancestor logic we have shown above. Other things which fall in the realm of LMP include:

- Verification of source code to some higher-level description (for example, conformance checking to coding conventions, design models, architectural descriptions, etc.).
- Extraction of information from source code (for example, visualisation, software understanding, browsing, generation of higher-level models or documentation, measurements, quality control, etc.).
- Transformation of source code (for example, refactoring, translation, re-engineering, evolution, optimization, etc.).
- Generation of source code.

Examples of these can be found in the research by De Volder on type-oriented LMP and TyRuBa. [Vol98] There is also the work by Wuyts on SOUL, which integrates LMP with a Smalltalk environment. [Wuy01] Kim Mens applied LMP for automating architectural conformance checking, and intentional views. [Men00, MMW02a] Tom Mens and Tourwé did research on evolving design patterns on a higher level by using LMP. [MT01]

1.2.3 Aspect-oriented programming

Aspect-orientation (AO) is a relatively new paradigm, which has grown from the limitations of Object-orientation (OO). [Kic97] OO takes an object-centric view to software development, where a programmer describes objects, how they should behave and in what ways they should interact with other objects.[2]

The problem with OO is very simple: *some concepts cannot be cleanly captured in an object*. Consider for instance logging —which has become *the* hallmark of AO. All objects and actions needing to be logged have to actively participate in order to achieve this goal. Any implementation of logging will therefore be spread out (or *scattered*) over the entire

[2] While typically achieved by constructing a hierarchy of classes, we should point out that OO is not limited to class-based approaches.

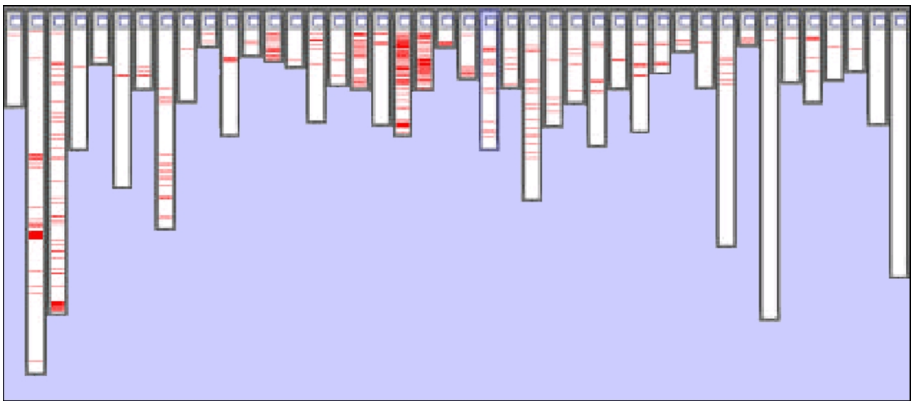


Figure 1.1: Logging in Apache Tomcat.

application. Figure 1.1 shows this for Apache Tomcat (a webserver implemented in Java): each column represents a class, with red horizontal bars indicating the presence of the logging concern. As can be seen, the logging concern is scattered over the application. It is not even in a small number of places. It is evident that such scattered concerns are hard to implement, and harder still to maintain.

A more extensive list of similarly hard problems can be compiled when working through AOSD resources:

> logging, tracing, context-sensitive error handling, coordination of threads, remote access strategies, execution metrics, performance optimisation, persistence, authentication, access control, data encryption, transaction management, pre-/post-condition and invariant checking, enforcement of policies for resource access and API usage, implementation of design patterns, test-coverage analysis,...

AO proposes the concept of *aspects* to solve the problem. Aspects allow one to *quantify* which events in the flow of a program are of interest (through so-called *pointcuts*), and what should happen at those points (through *advice*). Hence we can 'describe' what logging means to an application and have the *aspect-weaver* (a compiler for aspects) take care of the hard and repetitive bits for us.

Filman and Friedman summarize it as follows:

> *"AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quan-*

> *tifications hold over programs written by oblivious programmers.*" [FF05]

As an aspect is a (module of a) program that operates on other (modules of) programs, we can say that AOP fits our definition of meta-programming. This is put more strongly by Steimann, who states that:

> "*there can be no aspect without a meta-level.*" [Ste05]

It is precisely through the *quantification* mechanism that this meta-level exposes itself in AOP. And it is here that we have an opportunity for a synthesis between AOP and LMP (explained in the previous section). In fact, it has been argued that more than an opportunity, it is probably the best approach to quantification. [GB03, OMB05]

Filman and Friedman's summary also points to another property of AOP: *obliviousness*. While the extent to which this has to be achieved, or whether it can be achieved at all, is still under heavy debate, there is one side to obliviousness that seems accepted: the base language need not know about AOP for AOP to work. That is, any valid program in the base language can have aspects applied to it. This makes AOP directly applicable to existing applications.

1.2.4 Recording architectural models using LMP

We have seen that knowledge about applications is crucial for being able to evolve them. Without this, software developers can only handle change to the best of their abilities. This will likely involve code duplication, reinvention of designs, or even breaches of the existing architecture.

Knowing this, developers resort to reverse engineering before making modifications. [CC90, Tic01] Indeed, it is stated that:

> "*Software forward engineering and reverse engineering are* not *separate concerns.*" [CC90]

Any kind of forward engineering requires an understanding (or *model*) of the application. This entails a reverse engineering of that application when that model is missing.

Models recovered in this way will most likely fall short of the original architecture, as reverse engineering requires a huge intellectual effort: it takes many context switches and good concentration in order to

form, and keep, a picture of the design. And all this will likely have to be redone when the time comes for another modification.

It is therefore important that models which result from such reverse engineering efforts be recorded, so that these models can become online software documentation that can be exploited in subsequent activities. [Hon98] Declarative meta-languages (and, hence, logic meta-languages) offer a nice way to achieve this. In fact, this is exactly what Kim Mens argues for in his work on conformance checking:

> "*[LMP] is very well suited to describe the mapping of architectural concepts and relations to implementation artifacts and their dependencies.*" [Men00]

Also, in [DDMW00]:

> "*It seems intuitively clear that design information, and in particular architectural concerns, are best codified declaratively as constraints or rules.*"

That this is more than mere intuition has been shown by Wuyts and colleagues in their work on SOUL, or the *Smalltalk Open Unification Language*. [Wuy98, DMW99, Wuy01, MMW02b] There is, for instance the excellent work on Intentional Views. [MMW02a]

Furthermore, Chikofsky and Cross state that being able to generate different views is a key objective for reverse engineering. [CC90] Moise and Wong show an application of this idea in [MW03]. Again, in [Men00], Kim Mens shows that LMP:

> "*allows the definition of multiple, potentially overlapping, architectural views, thus providing support for separation of concerns at the architectural level.*"

It is for these reasons that Logic Meta-programming becomes a key technology for the work in this dissertation.

1.2.5 Exploiting architectural models using AOP

Models are a great way for representing knowledge about applications, which, as we have seen, is a basic requirement for helping these applications evolve. A good model becomes even better as soon as one can start to exploit it, not just at an architectural level, but also in code. In [MMW02b], the authors argue for:

> "*declarative programming as a basis for building sophisticated development tools that aid a programmer in his programming tasks.*"

StarBrowser[3], by Roel Wuyts, is a nice example of this idea put into practice. What's more, in [DFW00]:

> "*The resulting information can, for example, be used by meta-components to drive code generation and refactoring.*"

It is the idea that Aspect-oriented Programming can be used in the role of such a meta-component that is the driving force for our thesis. One argument in favour of AOP was made by Kim Mens, who notes that

> "*architectural descriptions, by their very nature, seem to cross-cut the implementation.*" [Men00]

In [DD99b], Maja D'Hondt and Theo D'Hondt argue that the same is true for domain knowledge. If so, then AOP, which is aimed at dealing with crosscutting concerns, should prove to be an invaluable tool in conjunction with models.

Another argument can be found in the work by Murphy and Notkin on lightweight source model extraction. [MN95] Part of their approach to extracting models is to (1) define patterns of interest in the source code, and (2) to define what action must be taken when such a pattern is found. Here is a slightly reformatted example found in their paper:

```
1 [<type>] <name> \( [{<arg>}+] \)
  [{<type> <decl> ;}+] \{
3
    <calledName> \( [{<par>}+] \) ( \) | ; )
```

Without going into the details, this matches procedures (pattern on lines 1 and 2) and the procedure calls they contain (nested pattern on line 4).

Defining the patterns of interest is the declarative bit, similar to quantification in AOP. It is not a full DMP approach, as their pattern matching language is restricted to EBNF-like notations. While this is a limitation in the context of this dissertation, it has distinct benefits for the work of Murphy and Notkin: it is simple and very tolerant of syntactical deviations in source code.

[3] http://homepages.ulb.ac.be/~rowuyts/StarBrowser/

Defining the actions to take when a pattern is found is close to the concept of advice in AOP. The main difference is that AOP-like advice is woven into the application to act at run-time, whereas the action code used by Murphy and Notkin acts during matching (what might be tentatively called weave-time). Here is an example:

```
  [<type>] <name> \( [{<arg>}+] \)
2 [{<type> <decl> ;}+] \{

4   <calledName>
      @ write ( name, " calls ", calledName ) @
6   \( [{<par>}+] \) ( \) | ; )
```

This again matches all procedures definitions and the procedure calls they contain. For every embedded procedure call the name of the procedure doing the call (`name`, matched on line 1) and the name of the procedure being called (`calledName`, matched on line 4) is output (`write` statement on line 5) during the matching.

Whatever the limitations are, the point is this: a combination of declarations on the structure of a program, combined with actions to take on the results thereof provided a practical, flexible and useful tool for developers.

One more argument can be found in an experiment using the SOUL environment. SOUL, or Smalltalk Open Unification Language, provides a framework that uses Prolog to reason about the structure of Smalltalk programs, as well as interact with those programs. In [DMW99] the authors use this in an AOP-like approach to separate domain knowledge from a path-finding algorithm. In fact, they go quite a bit beyond AOP as the Prolog language will engage the Smalltalk program at run-time as well. While this grants the authors greater expressivity and flexibility, this level of integration becomes hard to achieve when dealing with legacy business applications written in languages lacking the meta-programming features present in Smalltalk.

All in all, AOP provides a practical tool for allowing developers to exploit architectural models, which in itself provides a catalyst for the development and recording of such models.

1.2.6 Recovering architectural models using AOP and LMP

In the previous sections we have argued that Declarative Meta-Programming, in conjunction with Aspect-oriented Programming, can help with forward engineering. It can however also provide the basis from which to do reverse engineering, and from which to build up our models of the software which is to be evolved.

We do not advocate an approach wherein reverse engineering techniques and toolchains are (re-)implemented using DMP/AOP, though there is some point to that. The SOUL framework, for instance, has been used to extract design patterns present in Smalltalk applications. [Wuy98] Murphy and Notkin used a lightweight declarative approach to mine for source models. [MN95] The work on Intentional Views demonstrates this as well. [MMW02a]

Rather than reworking reverse engineering techniques to fit the DMP/AOP approach, it should be easier to use DMP/AOP in order to *enable* such techniques. Information on program structure can be extracted using DMP. Information on its run-time behaviour can be retrieved using aspects. An example of the latter will be shown in chapter 5, where we enabled the frequency clustering technique found in [ZD04] through a simple tracing aspect.

The argument here is not that DMP/AOP is the ideal tool to do reverse engineering. The argument is that DMP/AOP can provide a uniform medium from which to start using these techniques without having to integrate them directly into the runtime environment.

1.3 Hypothesis

We have seen that business applications, which are instantiations of specific business processes, are highly susceptible to the evolution of these processes. With increased globalization of enterprises, and ever greater demand for interconnectivity between companies, comes increasing pressure to scale up and integrate business applications. This means that these applications will have to evolve. Aside from difficulties in integrating the different business *models* and their associated business *processes* (which is well outside the scope of this dissertation), getting the business *applications* to cooperate is a major hurdle: in all but a few cases the documentation and support of these applications is insufficient (or even absent). Yet, in order to get applications to evolve

in a manageable way, information about its structure and behaviour is required.

When we considered this required information in the form of models, we found that their recording and exploitation is important. Furthermore, we have indications that, in this respect, Logic Meta-programming and Aspect-oriented Programming can help us out. This idea, put more strongly, is the thesis of this dissertation:

Hypothesis. *The combination of Logic Meta-programming and Aspect-oriented Programming aids in the recovery of business architectures from source code, as well as in the restructuring and integration of business applications.*

We make this claim based on two observations. First, by embedding Aspect-oriented Programming in existing business environments we can empower software developers with a flexible toolchain while avoiding a steep learning curve. In using this toolchain, there is no requirement to move away from the existing development techniques; there is only the incentive to work with something that augments them. This can make for a faster turn-around based on available expertise.

Second, Logic Meta-programming (a form of Declarative Meta-programming) can be used for expressing business concepts and architectural descriptions of business applications in a declarative way. This makes it possible to work with applications at a higher level of abstraction, which will allow better architectural descriptions to emerge. By making these descriptions available for practical use we can actively encourage development and understanding thereof.

1.4 Approach

Testing of our hypothesis requires the availability of Aspect-oriented extensions to programming languages that are in use for business applications today. For this we choose Cobol (accounting for over 80% of such applications) and ANSI-C. We choose these languages because they represent two very different approaches to structuring programming. By showing that our hypothesis will work in these two languages, we can thus be assured that our approach can reasonably be

applied to most other languages.

At the start of our research there were no readily available Aspect implementations for either C or Cobol, which meant that our primary focus had to shift to extending these programming languages with Aspect-oriented constructs. This became the major brunt of our work, and will account for most of this dissertation.

Apart from the addition of the aspectual constructs themselves, we then extended these constructs themselves by embedding LMP. This requires the definition of bindings and backtracking semantics to the quantification mechanism. We discuss how this was done.

Apart from the definition of (the semantics of) these extensions, we also discuss how to actually implement this in tools. The toolchain which was developed for this is based on XML representations of the abstract syntax trees of the source code. We discuss the reasoning behind this choice, as well as how the toolchain was built around it.

Using the resulting tools we then present several scenarios, some on restructuring and integration, some on recovery, to show how AOP and DMP can help. We also report on an industrial recovery case, which has been performed in cooperation with the LORE[4] research lab from the University of Antwerp.

1.5 Contributions

Summarizing, the main contributions of this dissertation are:

1. We show that the Aspect-oriented Paradigm can be successfully embedded in legacy (non-OO) environments.

2. We show that logic-based pointcuts, in conjunction with access to a meta-"object" protocol, are able to fully enable Aspect-oriented Programming in legacy environments lacking reflective capabilities.

3. We argue that Aspect-oriented Programming, together with logic-based pointcuts, is an ideal tool for tackling major issues in business applications: recovery, restructuring, integration, evolution, etc.

[4] *Lab On REengineering* (`http://www.lore.ua.ac.be/`)

1.6 Organisation of the dissertation

The organisation of this dissertation is as follows. Chapter 2 takes the first step toward proving our hypothesis by enabling Aspect-oriented Programming in a legacy language, namely ANSI-C. The choice for C is a pragmatic one: it is found within many business applications and presents a non-Object Oriented approach to programming. It is also greatly different from Cobol, which is the language focussed on in the next chapter. We discuss the different steps to take for adding AOP to ANSI-C, as well as how the weaving process can be implemented.

Chapter 3 takes the next step by enabling Logic Meta-programming in a legacy language. The choice is for doing this in Cobol, which is the language found most in legacy contexts. It is also very different from C, so that we must re-evaluate the addition of AOP constructs to the language. This will also revalidate our approach from the previous chapter. Furthermore, we show how LMP can be enabled within the pointcut language of the AOP extension, as well as what impact this has on the semantics thereof.

Chapter 4 then presents the transformation framework which was used to implement the aspectual extensions from the previous chapters. The framework itself is based on XML representations of the abstract syntax trees of the legacy source code, and we discuss the reason for this choice. We also present how the framework may be used in different ways, using different technologies and for different purposes.

Having seen how AOP+LMP can be designed and implemented with respect to different legacy languages, chapter 5 explores what these tools can do for us. We show how AOP and LMP can be fully enabled in ANSI-C, thereby completing the discussion of chapter 2 with the insights from chapter 3. Furthermore, we present how the transformation framework from chapter 4 was applied to source code visualisations, thereby showing the flexibility of our this framework. Most importantly, we discuss the application of AOP and LMP to several, very different problems with legacy software, thus arguing in favour of our hypothesis. We do this in the context of re-engineering, restructuring and integration of legacy applications.

Finally, in chapter 6, we round up and present our conclusions.

Chapter 2

Enabling AOP in a legacy language: ANSI-C

C is quirky, flawed and an enormous success.

DENNIS M. RITCHIE

OUR first step toward proving our hypothesis is to enable Aspect-oriented Programming in a legacy language. Without this ability, the question of whether AOP and Logic Meta-programming can help with the ills of legacy software would become moot. Hence this first chapter. The choice for C is a pragmatic one: it is found within many business applications and presents a non-Object Oriented approach to programming. It is also greatly different from Cobol, which is the language we will focus on in the next chapter on embedding LMP. Between enabling AOP+LMP in C and Cobol we have shown that our approach scales to a wide range of programming languages.

2.1 A brief history of C

The C programming language started life as an extension, by Dennis M. Ritchie, of the B programming language. [Rit93, BG96] B was itself a rewrite of a still older language, BCPL, in an attempt to fit it into 8KB of memory (while also reworking some of its features). Though successful, the machines running B were too small and too slow to allow more than mere experimentation.

In the early seventies Ricthie extended B by adding a character type, and also chose to rewrite its compiler to generate PDP-11 machine instructions instead of threaded[1] code. This resulted in a compiler capable of producing small enough programs which could compete with their hand-coded assembly counterparts. He dubbed this new version NB, or "new B".

Ritchie continued work on NB by extending the type mechanism and adding support for structured (record) types. He allowed these record types to be constructed from other record types, rather than restricting their definition to the basic primitives (`int` and `char`, together with arrays of them, and pointers to them). After this came a final renaming, in which Ritchie decided to follow the single-letter style, leaving open the question of whether the name represented a progression through the alphabet or through the letters in BCPL.

The language and compiler were now strong enough to rewrite the Unix-kernel for the PDP-11. The compiler was subsequently retargeted to other machines (the Honeywell 635, IBM 360/370,...), and libraries were being developed (among them the "Standard I/O routine").

In 1978, Kernighan and Ritchie published *The C Programming Language* [KR78], which became the de-facto standard for C programming. This standard remained in effect until 1989, when the X3J11 committee, established by ANSI, produced the first ANSI-C standard. [Ame89]

2.2 Existing aspect languages for C

While there now exist several different aspectual extensions to C (Arachne [DFL+05], TOSKANA [EF05], TinyC [ZJ03]), at the time when our own work started there really were only two options: AspectC by Coady et al., or AspectC++ by Spinczyk et al.

2.2.1 Coady's AspectC

The first AOP extension to appear for C was (aptly named) AspectC. Developed by Coady et al., it was developed for a research project, in which they tried to improve the modularity of path-specific customizations in operating systems code. [CKFS01] The authors reported As-

[1] An interpretative scheme in which the compiler outputs a sequence of addresses of code fragments that perform the elementary operations.

pectC to be:

> *"a simple subset of AspectJ [KHH+01]. Aspect code, known as advice, interacts with primary functionality at function call boundaries and can run before, after or around the call."*

Where at that time AspectC code was still hand-compiled to native C, in a follow-up paper the development of a prototype weaver is reported. [CK03] Unfortunately, we could not get this weaver up and running in our own environment, which made further evaluation difficult.

2.2.2 Spinczyk's AspectC++

The second option for an existing tool is Spinczyk's AspectC++, an aspectual extension of C++. [SGSp02, LS05b] While in theory you might be able to tackle C code using this tool as well (C++ originally being an extension on C), in practice this is not (yet) supported. The developers of AspectC++ do have support for C in mind, but, according to the FAQ published on their website[2], preliminary support was only to be available from January 2006.

2.3 Wicca

Wicca[3] is the name for our[4] aspectual extension of ANSI-C. It features a pointcut language which is close to that of AspectJ, but which has been tweaked for the context of ANSI-C. As with AspectJ, it also chooses to stay as close as possible to the original language. We do this so that users of the language may get up to speed with it more quickly, and need to do as few "context switches" as possible.

Wicca was tested in-house against some academic examples. It was also applied successfully by Tom Tourwé to some industrial C code (from ASML; see also section 5.1.2).

[2] `http://www.aspectc.org/`

[3] `http://allserv.ugent.be/~kdschutt/aspicere/`

[4] Developed with the help of Stijn Van Wonterghem, in the context of his graduation thesis [Won04] (in Dutch), with the author acting as tutor.

2.4 The join point model for Wicca

The goal of this chapter is to evaluate the possibility of enabling an existing (legacy) language with Aspect-oriented Programming. Now, the expressiveness of any AO programming language is defined first and foremost by the run-time events it can capture and work on. These events are, what is called in AOP, the *join points*. A description of which join points matter, forms the *join point model* of the language.

The join point model itself is totally reliant on the semantics of the base language to which the aspectual extension applies; it is a function thereof. When it comes to procedural languages (i.e. C) the possibilities for a join point model prove quite limited: events surrounding procedures. Note that this does not have to exclude access (reading and writing) to variables. These can be covered by equating such accesses to the use of some implicit getter and setter procedures, something which is allowed in C. In this way we can unify, and simplify, the join point model. One could argue for going more fine-grained than procedures (i.e. down to the statement- or expression-level). However, current best practice in AOP seems to suggest that this is not necessary. While we consider it a point open for debate, we chose to follow current practice; also because it simplified the problem domain.

Furthermore, we will limit ourselves to consider only *calls* of procedures. We do *not* make a distinction between call- and execution-join points as in AspectJ-like languages. Semantically speaking, there is no difference between the two: they cover the same points. From the point-of-view of the weaver, however, there are times when being able to make the difference is useful. Consider having to build an application while making use of a library which is only available to you in binary form. Unless binary weaving is an option, this means that weaving at execution-side is off limits. Having call join points can then solve the problem by forcing call-side weaving (at least partially; relevant calls within the library will still be discarded). The choice we made for Wicca was to place the burden of finding the most appropriate join point-weaving scheme (call- or execution-side) with the weaver. Given the choice, we prefer a simpler semantics and a smarter weaver over more complex semantics and a simpler weaver.

In summary, Wicca's join point model is based on an overlay of a message model on C, where messages are aligned with the activation of procedures. This approach can be applied to any procedural language.

2.5 Join point selection in Wicca

Given a join point model for a legacy language, the next most important factor in the definition of an aspect language is how we can select (quantify) the relevant join points. In AOP terms, this is the *pointcut mechanism*. With Wicca we decided to go for an AspectJ-like pointcut mechanism, as will be explored in the following subsections. This approach can be equally easy applied to other languages which allow for a message model.

2.5.1 Primitive pointcut designator

There is only one basic pointcut designator, and it is one which selects messages based on the message name. The syntax is:

```
on ( regexp )
```

This pointcut selects any message whose message name matches the `regexp` pattern (which is a standard regular expression, e.g. "`set.*`" would match any name starting with "set").

There are no pointcuts allowing selection based on argument- and/or return types in the original Wicca. These could, however, be added to the language. Selecting on return types can be done easily:

```
returning ( type )
```

A pointcut expression for checking for argument types is also straightforward:

```
args ( type )
args ( type, type )
args ( type, type, type )
/* And so on */
```

2.5.2 Pointcut composition

Pointcuts can be combined using the standard logic operators in C (`&&`, `||` and `!`), in the same way as in AspectJ. The following presents the EBNF-grammar for this syntax:

```
pointcut ::= conj_pc ("||" conj_pc)*
conj_pc  ::= prim_pc ("&&" prim_pc)*
```

```
prim_pc  ::= "(" pointcut ")"
prim_pc  ::= "!" prim_pc
prim_pc  ::= "on" (regexp)
```

Having defined the syntax, and thus, implicitly, the precedence of the combination operators, we now turn to the semantics. First:

```
pc_a || pc_b
```

This disjunction will match a join point when either `pc_a` or `pc_b` matches. Similarly:

```
pc_a && pc_b
```

This conjunction matches only when both `pc_a` and `pc_b` succeed. And:

```
! pc
```

This negation will match only when matching `pc` fails. As a combined example, consider:

```
on (.*) && ! (on (main) || on (printf))
```

This will select all messages, except for those named "main" or "printf".

2.6 Advice in Wicca

Having defined a join point model for C, as well as the pointcut language for quantifying over the right join points, we are now ready to turn our attention to defining behaviour at join point sites. In AOP, this is done through *advice*. In Wicca, the syntax is as follows:

```
returntype advice pointcut { advice_body }
```

Hence, advice is identified through the `advice` keyword. This is followed by the pointcut definition, which will select the relevant join points at which the advice should activate. What should happen is defined by the advice body, which is written down in ANSI-C. So, for instance, we might write:

```
void advice on (print.*) {
  /* advice body goes here */
}
```

This captures calls to all procedures whose name starts with "print".

Such advice acts as *around* advice. That is, it replaces the join point on which it acts, and is made responsible for forwarding to the join point at the right time (through a `proceed` call). As such, the advice needs to handle the value returned by the original join point. It also needs to return a value itself, which means that its type must be declared. This is what `returntype` is for.

The choice for limiting advice to *around* is one of convenience regarding the implementation. However, as *before* and *after* advice can be seen as special cases of *around* (with the proceed call at the end or at the beginning respectively), this does not restrict expressivity. It only has an impact on ease-of-use.

The major limitation is having to declare the return type. Wicca does not provide any form of auto-casting or auto-boxing, and so the burden of getting the typing right is put squarely on the shoulders of the programmer. While it is true that by using *before* and *after* advice one does not have to suffer this return-type problem, it is equally true that not all advices can be refactored to using only these two variants (e.g. conditional execution of a join point would not be possible in this scheme). Hence, the support for around advice remains necessary.

2.7 Accessing context information in Wicca

Having extended C with the basics for writing aspects, there is still one thing we have to provide: access to the execution context. This is something which is needed in even basic tracing advice. For Wicca, we allow context information to be accessed through a special function, named `this_joinpoint`, which returns a structure representing the captured join point:

```
typedef struct _jp_info {
  char *name;
} _jp_info;
```

As can be seen from the definition, this structure only holds the name (that is, the message name, or the name of the selected procedure) of the activated join point. Access to return types, argument types, and argument values was not made available.

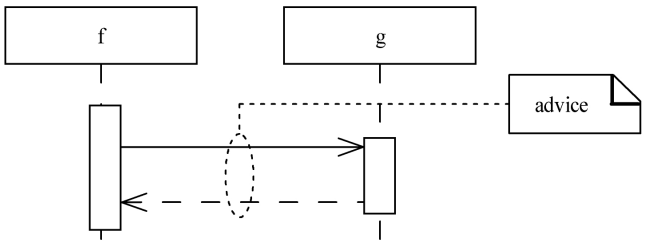


Figure 2.1: Initial situation.

2.8 Weaving Wicca applications

Sofar, we have only tackled the syntax and semantics of our aspectual extension to ANSI-C. There remains however the question of how this is transformed into executable form. This is what weaving is all about.

Weaving of advice on join points is based on the idea of *call chaining*. In short: we instantiate advices into procedures, and place them between the procedures participating in a call by redirecting those calls. This can be done easily in any procedural language.

2.8.1 The weaving process

We will now illustrate call chaining, to make the technique more clear. Consider two procedures, `f` and `g`, where `f` makes a call to `g` (figure 2.1). The message which is overlaid on this call/return sequence is advised by some aspect.

This advise will then be woven using the following steps:

1. Procedure `g` is renamed to `g'`, where `g'` is a new, unique name.
2. The advice is instantiated as a procedure with a signature matching that of the original `g`. This procedure is named `g`.
3. Any `proceed` calls in the instantiated advice are replaced by calls to `g'`.

This process is repeated for all advices acting on `g`. In case of a single advice, we end up with figure 2.2. When dealing with multiple advices, the order in which they are activated (or woven in) is dictated by the order in which they appear in the source code.

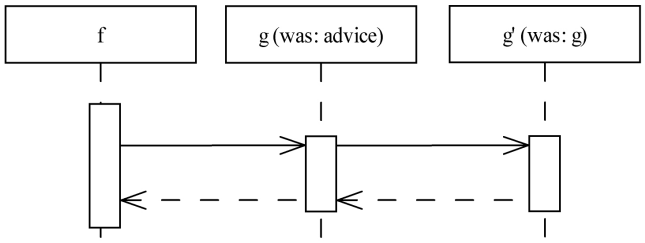


Figure 2.2: Advice chained in.

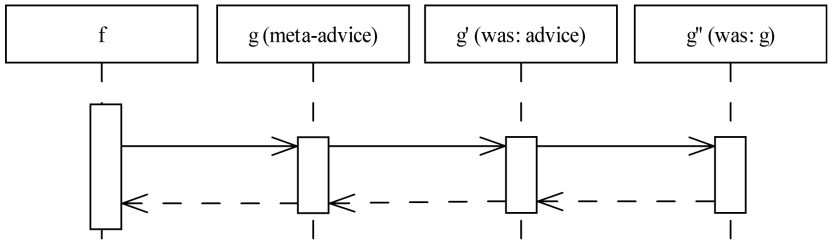


Figure 2.3: Meta-advice chained in.

The weaving is then rounded up with the addition of one more advice: a meta-advice which registers information on the currently activated join point (made accessible through `this_joinpoint`; see the previous section). This brings us to the end result of figure 2.3.

Note that, due to the way advice is woven into the application, we can also handle function-pointers. These will now simply point to the meta-advice at the front of the advice stack, as this meta-advice replaces the original procedure.

2.8.2 Limitation

The call chaining approach to weaving that has been described here is one which matches *execution-side* weaving. The assumption here is that this side is available to the weaver. When the sources are available, this is not a problem. However, when the execution-side is part of a pre-compiled library, we are forced to find other approaches.

As was stated in section 2.4, we expect the weaver to be able to choose between call-side and execution-side weaving as appropriate. This choice, however, is not a trivial one, as it requires knowledge of

the full build scheme. In C it is pretty common to first build separate object files, and only then link them all together. This means that at any given step, we may not yet know if the execution side is available to us. It may be that the execution side resides in an object file which is still to be built. Hence, for our weaver to take the right decision we would need to integrate it more closely into the build process. An alternative may be to go for a two-phase process where the first phase is for collecting all the needed information, and the second step is where the actual building takes place. But even this is hard when the build process has to take other tools into account. [ZAD+06]

Until we have delved more deeply into this subject matter, we decided to have Wicca always do execution-side weaving. Given the initial intent for Wicca (see section 2.3), this was a reasonable choice.

2.9 Evaluation and critique

While limited in its features, Wicca can already express some basic aspects. Here is an example:

```
1 int advice on (.*) && ! on (printf) {
    int r = 0;
3   printf ("before␣%s\n", this_joinpoint()->name);
    r = proceed ();
5   printf ("after␣%s\n", this_joinpoint()->name);
    return r;
7 }
```

This shows the, in AOSD-literature omni-present, tracing aspect. It captures all procedures, except for `printf` (line 1). This exclusion is needed so as to prevent an infinite loop, in which the advice would keep triggering itself over and over. It surrounds the original join point (activated through a `proceed` call at line 4) with trace-printing code (`printf`'s at lines 3 and 5). Line 1 declares a variable, which captures the value returned by the original join point (assignment at line 4), and is then returned (unchanged) as the result of the advice (line 6).

As an aside, restricting traces to message/procedure names is not unrealistic. In C there is no overloading of procedures allowed. This means that any procedure name can be tied to only one signature, and, hence, that every name maps onto one uniquely defined procedure.

The real problem with Wicca's advice structure, as it is presented here, is the need to declare return types. The above example works, but only on the assumption that all join points return integer values. This is only likely to be true in a few academic examples.

One possible solution to this problem is the addition of a pointcut selector which will discriminate on return type. For instance:

```
on (.*) && ! on (printf) && returning (int)
```

While this would make it safe to apply our advice to any code base, it does not solve the more essential problem: the tracing advice does not really care about the return types. On the contrary, it should be applicable to any and all procedures regardless of return type. Attaining this through a `returning` pointcut would mean (1) knowing all possible return types in advance, and (2) writing down the advice for every such return type. This is clearly more trouble than it is worth. What we would really like is something in the vain of:

```
1 any advice on (.*) && ! on (printf) {
    any r = 0;
3   printf ("before␣%s\n", this_joinpoint()->name);
    r = proceed ();
5   printf ("after␣%s\n", this_joinpoint()->name);
    return r;
7 }
```

I.e. we would prefer to use a generic type (`any`), and have the language take care of the necessary casting.

Logic Meta-programming provides us with a similar solution, without having to resort to hidden type casts. We will present this technique in the discussion on Aspicere (section 5.1).

The other pitfall is that Wicca will always do execution-site weaving (see the previous section), which means that our primitive pointcut (`on (...)`) is no more than an *execution* join point, as in AspectJ. This limitation, however, is caused by the integration of the weaver into the build process; *not* by the semantics of the join point model.

2.10 Conclusion

In this chapter we considered the addition of AOP constructs to an existing (legacy) language, in casu ANSI-C. We find that we can represent

join points of interest by overlaying them with a message model. Based on this we have then set up a pointcut language, and integrated it with the original language through the means of advice definitions. Adding AOP to an existing language turns out to be straightforward, and stays close to existing AOP approaches. It turns out however that our ability to write generic advice is hampered by the lack of reflection and a limitation of type conversions of the underlying language. As this is something we can rightly expect to find in most legacy languages, our solution sofar, while already effective, is still incomplete. The addition of LMP techniques in the next chapter will remedy this.

Chapter 3

Enabling LMP in a legacy language: Cobol

COBOL is a very bad language,
but all the others (for business data processing)
are so much worse.

ROBERT GLASS

SOFAR we have shown that an existing legacy language (in casu ANSI-C) can be extended with Aspect-oriented Programming. We have however found that this extension is limited in its use by the lack of reflective capabilities of the underlying language, as well as the limited support for type conversions. This chapter takes the next step by now also enabling Logic Meta-programming. This time the choice is for doing this in Cobol, which is the language found most in legacy contexts. It is also very different from C, so that we must re-evaluate the addition of AOP constructs to the language. This also revalidates our approach sofar.

3.1 A brief history of Cobol

On May 28 and 29, 1959, a meeting was called in the Pentagon by Charles A. Philips of the Department of Defense. [Sam69, Sam85] It was attended by representatives from users, government installations, computer manufacturers, and other interested parties; a group which has

since become known as the CODASYL (*CO*nference on *DA*ta *SY*stems *L*anguages) committee. The goal of this meeting was simple: to discuss the problem of developing a common business language. The word *common* was interpreted to mean that source programs should be compatible among a significant group of computers.

The committee decided that such a project was both desirable and feasible, and so they set up a more thorough project. The first step was to be performed by the "Short Range" committee, whose job was to do a fact finding study of what was wrong and right with existing business compilers (FLOW-MATIC, AIMACO, COMTRAN,...). This committee, however, decided to go for the more ambitious goal of specifying a language instead.

The Short Range committee did so, and presented its work on September 4 of the same year. The language it presented was quickly adopted by CODASYL, and was baptised Cobol, or *CO*mmon *B*usiness *O*riented *L*anguage. In the subsequent years Cobol went through several standardization phases, though this did not prevent the appearance of many dialects. Its popularity is undeniable: in 1997 the Gartner Group estimated that Cobol accounted for over 80% of all running systems. Recent statistics for 2004-2005 still estimate the use of Cobol for new software at 15%, where 80% of all applications will have to co-operate in some way with existing legacy programs. At the age of 46 this makes the language very much alive-and-kicking.

This chapter won't delve into Cobol specifics. For an introduction to Cobol, albeit a brief one, we refer the reader to appendix A.

3.2 Existing aspect languages

While there exist no specialized aspect extensions for Cobol, Cobol itself is perhaps the oldest Aspect-oriented programming language. The procedure division of a Cobol program can hold so-called *declaratives* with special `USE`[1] statements, which provide a method of invoking procedures that are executed when some distinguished condition occurs during program execution. I.e., Cobol allows the association of behaviour to some quantified events. This is, in essence, a form of AOP. The following fragment shows a typical example of this.

[1] The `USE` verb is in all Cobol standards, e.g., in the Cobol 74 standard. [Ame74] It was already present in some form in CODASYL's Cobol 60. [Sam78]

```
1  DECLARATIVES.
     HANDLE-F0815-ASPECT SECTION.
3     USE AFTER ERROR ON FILE-F0815.
     HANDLE-F0815-ADVICE.
5     MOVE "F0815" TO PANIC-RESOURCE.
      MOVE "FILE␣ERROR" TO PANIC-CATEGORY.
7     MOVE FILE-STATUS TO PANIC-CODE.
      GO TO PANIC-STOP.
9  END DECLARATIVES.
```

Without going into too much detail, this code makes sure that each file I/O error related to `FILE-F0815` is caught (line 3; *quantification*). Paragraph `HANDLE-F0815-ADVICE` (lines 4–8; *advice*) encodes what should subsequently happen: registration of the error, followed by a jump to the `PANIC-STOP` procedure, which will display a decent error message and then stop execution.

From the viewpoint of AOP, the incompleteness of Cobol's accidental pointcut language is evident: we cannot advise *successful* file I/O statements, subprogram calls and field access. Also, Cobol's accidental join point control and join point reflection are very limited. For instance, we cannot re-execute an offending statement within a handler section, which hampers error repair. As such, Cobol's "AOP mechanism" is not able to tackle arbitrary aspects. Its design is simply too specialized to allow this.

3.3 Cobble

Cobble[2] is the name for our[3] aspectual extension to Cobol. It presents an extensive definition of AO constructs for Cobol, which is very different from other aspect languages. This is, of course, due to the very different nature of Cobol itself. As with AspectJ-like aspect languages, though, it also chooses to stay as close as possible to the original language. We do this so that users of the language may get up to speed with it more quickly, and need to do as few "context switches" as possible.

[2] Etymology: `http://dictionary.reference.com/search?q=Cobble`
Download: `http://allserv.ugent.be/~kdschutt/cobble/`

[3] Exploration of this design was done in cooperation with Ralf Lämmel (then: Free University & CWI, Amsterdam; at the time of writing: Microsoft, Redmond). Work started from a preliminary exploration done by David Tas for his graduation thesis ([Tas04]; in Dutch), with the author acting as tutor.

Cobble was tested in-house against some academic examples, using the OpenCobol[4] compiler.

3.4 The join point model for Cobble

We have seen in the previous chapter that any definition of an aspectual extension to a language must start by defining the join points of interest. Now, Cobol does not support the modern concept of procedures or functions when it comes to structuring control flow of programs. It does talk about some form of "procedures", but these are of a different form. Cobol relies on a structure which is close to handwritten texts: sections, subdivided into paragraphs, subdivided further into sentences, and again into statements.[5] As such, it follows pretty much the pattern of, say, a standard Ph.D. dissertation:

```
program   ::= section*
section   ::= identifier "SECTION" "."
              sentence* paragraph*
paragraph ::= identifier "." sentence*
sentence  ::= statement* "."
```

When we talk about procedures in Cobol, we are really talking about paragraphs and sections. From this we find that a message model would not match Cobol's structure, and so we must go for another approach.

In Cobol, new functionality (e.g. working with XML) is, in general, made available through the addition of new statements (or new 'clauses' in existing statements). The result is that a typical Cobol dialect will sport over 500 keywords. While it is possible to modularize applications into subprograms —Cobol programs may be called from other Cobol programs—, each of which is structured as described above, this is not very easy or practical. As such, the creation of APIs by programmers is not encouraged, and is left to the compiler constructors. Therefore, most events which may be of interest are available at the statement level. Hence, the basic join point Cobble exposes are executions of statements. Nevertheless, the structure provided for by paragraphs and sections is equally of importance, as they define the general

[4] http://www.opencobol.org/

[5] Also see appendix A for more details.

control flow. Cobble therefore exposes execution of paragraphs, sections and programs as well.

Now, statements can be divided further into *clauses*, but this depends entirely on the kind of statement. A `MOVE` statement, for instance, has a compulsary `TO` clause, whereas a `DIVIDE` statement has (among other things) an optional `REMAINDER` clause. Even more fine-grained we arrive at naming the data items (or variables). These can be used in one of two modes: *sending* or *receiving*. [ISO02, § 14.5.7; p. 389] For instance, x is a sending data item in `MOVE` x `TO` y, while y is a *receiving* one. Still, we choose not to go more fine-grained than the statement level. This is mostly for practical reasons: weaving clauses using a source to source technique is extremely hard —if at all possible. There is, for instance, no way to replace the reference to a data item with a call to a procedure. First, it is not allowed syntactically. Second, procedures cannot return values, and so cannot act as a data source.[6] We also disregard the concept of sentences as a form of join point, as we cannot identify any scenario where they might be of importance.

Concerning the distinguishing between call or execution sites, as with Wicca, Cobble does not do this. The rationale, again, is that there can be no difference of "sender" and "receiver" in Cobol: there exist no objects.[7] An argument for distinction of call and execution could be made for Cobol's subprograms mechanism. However, this case is already covered, as calls to subprograms are achieved through a special `CALL` statement. Because of our choice for exposing statements, we can already handle subprogram calls.

3.5 Join point selection in Cobble

Given the join point model for Cobol, the next most thing we must decide on is the definition of an pointcut language for the selection of the relevant join points. The pointcut mechanism for Cobble is decidedly different from that of AspectJ-like languages. For one, there is the focus on Logic Meta-programming, which has an impact on the expressiveness of the pointcuts. Then there is also the choice of syntax: "English", to follow Cobol's existing conventions.

[6] More recent Cobol standards do support the concept of "functions", but these are defined as a separate module, using the `FUNCTION-ID` keyword.

[7] While the latest Cobol standard does cover OO constructs (cfr. [Läm98]), our focus is on *legacy* business applications, which predate this.

3.5.1 Primitive pointcut designators

The most basic pointcut designators are those which pick out the individual join points: specific statements, (sub-) programs or procedures (sections and paragraphs).

```
jp ::= (verb | "ANY") "STATEMENT"?
jp ::= "PROCEDURE" procedure_name?
jp ::= "PROGRAM" program_name?
```

The nonterminal *verb* is a placeholder for Cobol's many statement verbs: `ACCEPT`, `ADD`, `ALTER`, ..., `WRITE`. Aspects for screen I/O deal with the verbs `ACCEPT` and `DISPLAY`. Aspects for file I/O deal with all the file I/O statements. Aspects for test coverage analysis deal with the verbs `IF` and `EVALUATE`. These aspects can specify which verbs are of interest to them, or choose `ANY` to capture them all.

We can also quantify over programs and procedures. By default all such constructs are matched. However, by naming them we can narrow the selection process down. E.g.:

```
PROCEDURE CALCULATE-PAYMENTS
```

This will only select paragraphs or sections named "CALCULATE-PAYMENTS".

By definition, we intercept the execution of a procedure using the `PROCEDURE` syntax. We also note that we can intercept procedures that are encountered 'by fall-through' rather than by `PERFORM` and `GO`, using the `PROCEDURE` syntax (for more details on how Cobol structures control flow see appendix A). More subtly, using the `STATEMENT` syntax, we can intercept (the execution of) the statement that performs a procedure (cf. the `PERFORM` verb) or jumps to it (cf. the `GO` verb). Likewise, we can intercept both (the execution of) a `CALL` statement and the execution of a subprogram.

We note that the syntax, given so far, does not yet accommodate constraints on the operands involved in the statements, neither can we access the parameters of intercepted subprogram executions. We will discuss such expressiveness shortly.

3.5.2 Pointcut composition

Pointcuts can be combined using the standard logic operators in Cobol (`AND`, `OR` and `NOT`), in the same way as in AspectJ. The following

presents the EBNF-grammar for this syntax:

```
pointcut ::= conj_pc ("OR" conj_pc)*
conj_pc  ::= term ("AND" term)*
term     ::= "NOT" term
term     ::= "(" pointcut ")"
term     ::= jp
```

Where *jp* refers to the definition in the previous subsection.

It is important to note that the logic deductions which can be specified using this syntax are based on *failure* and *backtracking*. That is, whenever a certain branch of deduction fails, the pointcut will backtrack to its previous branch-point and attempt another solution. This provides the basis for our Logic Meta-programming approach. The importance of this will become evident when we discuss the notion of ambiguous pointcuts (section 3.5.4).

Having defined the syntax, and thus, implicitly, the precedence of the combination operators, we now turn to the semantics. First:

```
pc_a OR pc_b
```

This disjunction will match a join point when either `pc_a` or `pc_b` matches. Similarly:

```
pc_a AND pc_b
```

This conjunction matches only when both `pc_a` and `pc_b` succeed. And:

```
NOT pc
```

This negation will match only when matching `pc` fails. As a combined example, consider:

```
EXECUTION OF ANY STATEMENT
AND NOT EXECUTION OF CALL STATEMENT
```

This will select (execution of) all statements, except for `CALL`s.

As an aside, the previous could also have been written as:

```
ANY AND NOT CALL
```

Optional keywords are very much a part of Cobol's flavour, and as such this example would not look out of place.

3.5.3 Join point conditions

AspectJ provides notational support for method-calling patterns, which resemble the syntactical structure of method calls, and which take advantage of the fact that method arguments are *typed* and *named*. In the case of Cobol, we face a much richer syntax, weaker typing, non-flat argument lists in subprogram calls, anonymous operands in statements and other complications. We have found that *selector-based access to program contexts* is more appropriate for Cobol than a plethora of patterns.

Selector-based access starts from the *join point shadow*[8], which must be a statement, a procedure or a program, and allows querying of its context. A selector's *applicability* depends, of course, on the category at hand. For instance, an attempt to extract a name is only valid when we face a *named* entity. Here are Cobble's selectors:

- `NAME` – The alphanumeric literal for the name of the selected entity.
- `VERB` – The alphanumeric literal for the statement verb at hand.
- `TYPE` – The type of a selected data item (a picture string).
- `PROCEDURE` – Navigation to the hosting procedure.
- `PROGRAM` – Navigation to the hosting program, i.e., to the root.
- `LEVEL 01` – Navigation to the hosting top-level data entry.
- `FILE` – Selection of the file in a file I/O statement.
- `RECORD` – Selection of the record in a file I/O statement.
- `FILE-STATUS` – Selection of the file-status field for the selected file.
- `SENDER` – Selection of a sending data item.
- `RECEIVER` – Selection of a receiving item.
- `PARAMETER` – Selection of a parameter for a subprogram or a call.
- `LOCATION` – The location in the source-code file.

[8] Where a join point is an event in the dynamic call graph a running program, a join point shadow is the code responsible for this event. [HH04]

- `IDREF` – An opaque identity —e.g. a unique number— for referring to the selected entity.

The list clarifies that there are selectors for the *extraction of basic properties*, such as names or references, and there are other selectors, which cater for the *navigation from one program context to another*. For instance:

```
NAME OF FILE-STATUS OF FILE
```

This will navigate from a file being accessed to the data-item holding the I/O status, and then to the name of that data-item.

We note that some selectors can be ambiguous. For instance, a statement can involve several sending items. We will handle such ambiguous selection in the next section.

Selection can be nested as in:

```
selector OF ... selector OF SHADOW
```

The final "`OF SHADOW`" can be omitted.

The result of selections can be bound to variables:

```
BIND var_name TO selection
```

These can then form the target of new selections:

```
selector OF ... selector OF var_name
```

As an example, the following binds `THE-PROC` to the procedure hosting the join point shadow:

```
BIND THE-PROC TO PROCEDURE OF SHADOW
```

This can then be used to, for instance access its name:

```
NAME OF THE-PROC
```

Or we can ask for its location in the source code:

```
LOCATION OF THE-PROC
```

And so on…

The list of selectors, as given above, is incomplete. Additional selectors can be derived systematically from the Cobol grammar. The difficulty lies in finding the right amount of details to be exposed to the Cobble programmer. E.g. do we expose details on individual arithmetic statements (`ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE` and `COMPUTE`),

or do we hide this behind the generic concept of a computation? In its current form Cobble does the former. When there is a need for working with the generic concept this can be added to the language as another selector.

Apart from the selectors, we also need a few condition forms that perform *classification tests* on a given syntactical entity, e.g.:

- x `IS FILE-DATA` – x is a data item declared in the file section.
- x `IS WORKING-STORAGE-DATA` – x is a data item declared in the working-storage section.
- x `IS LINKAGE-DATA` – x is a data item declared in the linkage section.
- x `IS SENDER` – x is a sending item.
- x `IS RECEIVER` – x is a receiving item.
- x `IS LIKE` *regexp* – x is an alphanumeric matching the pattern in *regexp*.

Put together, we can now write some very expressive pointcuts. Here is an example:

```
1 ANY STATEMENT
  AND BIND VAR-ITEM TO SENDER
3 AND VAR-ITEM IS FILE-DATA
  AND NAME OF FILE OF VAR-ITEM IS "CLASS-RECORDS"
```

This will capture execution of any statement (line 1) which reads data (test for `SENDER` on line 2) from a variable ("VAR-ITEM" on line 2) which is part of a record in a file (condition on line 3) named "CLASS-RECORDS" (test on line 4). As is evident from this example, Cobble's design strongly suggests that join point reflection on the join point shadow should be viewed *as part of the pointcut* — as opposed to using reflection in the advice code.

3.5.4 Ambiguous pointcuts

Some of the selectors presented in the previous section are ambiguous. Consider `SENDER`. The following `ADD` statement has three sending data items:

```
ADD A B TO C.
```

I.e., `A`, `B` and `C` are all variables from which data will be read. This means that the following selection will have three possible solutions:

```
BIND VAR-ITEM TO SENDER
```

This ambiguousness is *not* a bug. It is a *feature*. Remember from section 3.5.2 that the pointcut matching mechanism is based on failure and backtracking to choice points. It is exactly through the ambiguous selectors that these points of choice are established. Consider again the example from the previous section:

```
1 ANY STATEMENT
  AND BIND VAR-ITEM TO SENDER
3 AND VAR-ITEM IS FILE-DATA
  AND NAME OF FILE OF VAR-ITEM IS "CLASS-RECORDS"
```

The selection on line 2 will bind `VAR-ITEM` to one of the sending data items in our `ADD` statement. When the tests on lines 3 and 4 succeed, this solution is accepted and the pointcut is matched to the join point. When any one of the tests fail, the matching will backtrack to the selection on line 2 and attempt another binding for `VAR-ITEM`. This will repeat itself until all possible bindings have been exhausted, at which time we can conclude that the pointcut could not be matched to the join point.

Of course, given the ambiguous selectors, there may be several valid ways of matching a certain pointcut, each of which may present different solutions for the bindings. What happens in such a case is related to the way in which we access the join point context, which will discussed in section 3.7. First, we turn our attention to the structure for advice.

3.6 Advice in Cobble

Given a join point model and a pointcut language for picking out join points, we must now consider how to embed this into the underlying language. Our approach here is based on Cobol's `DECLARATIVES` section (see section 3.2). In the design of Cobble we have chosen to stay close to Cobol, so that both can integrate as seamlessly as possible.

3.6.1 Structure

The overall structure of an advice looks as follows:

```
    MY-ADVICE SECTION.
2    USE AFTER
       *> The pointcut designator.
4    MY-ADVICE-BODY.
       *> The advice code.
```

This declares an advice named "MY-ADVICE", which is an *after* advice (*before* and *around* are also allowed). The pointcut is declared inside a `USE` statement —generalizing existing uses for this statement. The expected behavior at the selected join points is encoded in one or more paragraphs following the `USE` statement (here: "MY-ADVICE-BODY"). The programmer is, of course, free to choose any names he wants. In the case of around advice, the original join point can be invoked through the `PROCEED` statement. The following shows a more complete example.

```
1   MY-ADVICE SECTION.
    USE AROUND
3     EXECUTION OF PROCEDURE.
    MY-ADVICE-BODY.
5     ADD 1 TO DEPTH.
      DISPLAY DEPTH, ": before procedure.".
7     PROCEED.
      DISPLAY DEPTH, ": after procedure.".
9     SUBTRACT 1 FROM DEPTH.
```

This is, in essence, a primitive tracing aspect. It is an *around* advice (line 2) which captures all procedure executions (line 3). The advice itself surrounds the original join point (invoked through a `PROCEED` at line 7) with some printing statements (`DISPLAY`s at lines 6 and 8), showing the current depth of the call stack (variable `DEPTH` which gets updated at lines 5 and 9).

3.6.2 Scope

The scope of Cobol's original declaratives is restricted to the hosting program. Our advices therefore resemble *intra-program* aspects. However, Cobble must also deal with *inter-program* aspects, as crosscut-

ting concerns are unlikely to align with subprogram boundaries. Inter-program aspects affect some or all Cobol programs in a given project.

Cobble provides a new form of compilation unit for inter-program aspects, which complements the pre-existing forms for programs and classes. The inter-program version requires a compilation unit as follows:

```
IDENTIFICATION DIVISION.
 ASPECT-ID. ASPECTS/LOGFILE.
```

This is similar to the `CLASS-ID` keyword for defining classes, as well as its `FUNCTION-ID` keyword for defining special functions, which are present in today's Cobol standard.

An aspectual compilation unit applies to all programs in a project. Of course, a program is *affected* only in case its execution exhibits relevant join points. Still, restricting this applicability to certain files is already possible using the given constructs. For instance, to prevent a logging aspect from logging its own behaviour we could write:

```
NOT NAME OF PROGRAM IS LIKE "ASPECTS/LOGFILE"
```

This idiom mimics AspectJ-like "within" pointcuts, which restricts join point selection to join points in a certain file.[9]

The environment and data divisions of the aspectual unit extend the affected programs. By default, Cobble separates the name-spaces of aspectual unit and affected program. Thereby, we avoid unintended name capture.

3.7 Context information

Again, advice often needs access to the execution context in which it is being activated. In Cobble it can do this by making use of the bindings that have been made in the pointcut (see section 3.5.3). The following subsections will elaborate on this.

3.7.1 Structure

Consider the following pointcut:

[9] More details can be found at `http://www.eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html`, under the header "Program text-based pointcuts".

```
EXECUTION OF PROCEDURE
AND BIND PROC-NAME TO NAME OF SHADOW
```

The first line selects execution of any procedure. The second line binds the name of this procedure to a variable named "PROC-NAME". Again, this could have been written more succinctly as:

```
PROCEDURE AND BIND PROC-NAME TO NAME
```

The added effect of this binding is that the variable which has been bound is now available for (read-only) use in advice. E.g., we could now write a tracing aspect which is able to output the names of the activated procedures:

```
1  MY-ADVICE SECTION.
    USE AROUND PROCEDURE
3     AND BIND PROC-NAME TO NAME.
    MY-ADVICE-BODY.
5     DISPLAY "Before ", PROC-NAME.
      PROCEED.
7     DISPLAY "After ", PROC-NAME.
```

As expected, the variable PROC-NAME which gets bound (at weave-time) on line 3, is available for use (at run-time) in the advice (lines 5 and 7).

3.7.2 Ambiguous context

Some selectors are ambiguous (see section 3.5.4): they return one of a set of possible results. Consider the following example of an aspect that counts certain sending operands in ADD and SUBTRACT statements:

```
1  USE BEFORE (ADD OR SUBTRACT)
     AND BIND VAR-ITEM TO SENDER
3    *> Disregard the item if it
     *> is a receiving data item, too.
5    AND NOT VAR-ITEM IS RECEIVER.
   MY-SENDER-ADVICE.
7    *> Count the sending data item
     *> if it equals zero.
9    IF VAR-ITEM = ZERO
       ADD 1 TO COUNT-ZERO-ITEMS.
```

Before any addition or subtraction (line 1), this aspect adds 1 to a counter (line 10) when the value of VAR-ITEM, a variable being read from (line 2) but not being written to (line 5), is zero (line 9). Putting this together, we see that this aspect detects superfluous additions and subtractions. Now take a look at the following addition:

```
ADD A B TO C.
```

This statement has two sending items which are not receiving data items: A and B. As such, this means that the variable referred to inside the advice (VAR-ITEM on line 9) can be either one.

Cobble solves this by assuming a universally quantified ('for all') semantics for the pointcut. That is, advice is issued for *all successful pointcut evaluations*, where ambiguous selections give rise to multiple solutions for the capturing BIND phrases. For the case of the above addition, this will be preceded by two executions of MY-SENDER-ADVICE; one for data item A and another one for B.

One might argue that some multi-valued selections can be avoided once the join point model of Cobble provides corresponding categories of join point shadows. (In the example, we would then formulate a pointcut on data items rather than on statements.) However, as was discussed in section 3.4, we disfavour this option. For instance, how to handle AROUND advice for access to data items? Advice, as it is based on paragraphs and sections, cannot return values, and so cannot act as a data source. Nor could a data item be replaced by a call to either advice or a procedure: Cobol does not allow this. Trying to get around this restriction while still generating legal Cobol code is a mind-boggling exercise, where statements must be split up (and be duplicated) in such a way that accesses to data items become separated. Here is a basic example:

```
IF A EQUAL TO 1 OR B EQUAL TO C THEN
  *> code ...
END-IF
```

In order to separate accesses to data items A, B and C, this would have to become:

```
IF A EQUAL TO 1 THEN
  *> code ...
ELSE
  IF B EQUAL TO C THEN
    *> code ...
```

```
      END-IF
    END-IF
```

Already we are seeing code duplication: the body of the original if-statement has to be cloned for all possible cases. Now consider that this body itself may also contain conditional statements. This would lead to an exponential explosion of code. Of course, we would not have to write this down ourselves; the weaver should take care of this. But then consider that Cobol has many more conditional statements than the simple `IF` statement. There are also the control flow statements which have to be taken into account. Not to mention the plethora of many other statements with their optional clauses and their own very different semantics. Each of these cases should be treated by the weaver. While we cannot prove this task to be an impossible one, we consider the complexity of attempting this to require extensive research on its own.

3.7.3 Metadata

One other type of context information comes in the form of metadata. This can be used to encode business information, or other things of interest. For instance:

```
1 META-DATA DIVISION.
  FACTS SECTION.
3   PARA-OF-INTEREST VALUE BIZ-AAB.
    PARA-OF-INTEREST VALUE BIZ-CDD.
```

This defines `PARA-OF-INTEREST` as a piece of metadata, whose value may be either `BIZ-AAB` or `BIZ-CCD`. These are then available for use within the pointcuts. E.g.:

```
  USE BEFORE (ADD OR SUBTRACT)
2   AND BIND VAR-ITEM TO SENDER
    AND PARAGRAPH EQUAL TO PARA-OF-INTEREST
4   AND NOT VAR-ITEM IS RECEIVER.
  MY-SENDER-ADVICE.
6   IF VAR-ITEM = ZERO
      ADD 1 TO COUNT-ZERO-ITEMS.
```

This is a reprisal of the aspect detecting spurious arithmetic from the previous section, only this time its scope is limited to certain para-

graphs (line 3). Exactly which paragraphs these are, comes from the metadate we defined previously.

3.8 Weaving Cobble programs

Sofar we have only tackled the syntax and semantics of Cobble. We must now face the question of how this functionality will get enabled in the woven application. We do this by illustrating how advice is woven in Cobble using the following intra-program aspect:

```
1  *> Print file name before file is read.
     USE BEFORE READ
3    AND BIND VAR-NAME TO NAME OF FILE.
   FILE-ADVICE.
5    DISPLAY "Reading␣", VAR-NAME, ".".
```

We are about to weave advice for this statement:

```
READ ORDER-FILE.
```

The most basic weaving scheme starts as follows: the advice is *cloned* per join point shadow (or source code responsible for activating a join point [HH04]), and placed within a newly created section. The cloned version is also subjected to substitution such that all variables (i.e. `VAR-NAME` on line 5) are replaced by the values that were obtained by pointcut evaluation (binding on line 3). Weaving advice for the sample `READ` statement, we obtain the following clone:

```
AOP42-FILE-ADVICE SECTION.
FILE-ADVICE.
  DISPLAY "Reading␣", "ORDER-FILE", ".".
```

The name, "AOP42-FILE-ADVICE", is only meant as an example. It is generated by the weaver. The final step is to extend the actual join point shadow by a `PERFORM` statement such that the cloned paragraph is executed:

```
PERFORM AOP42-FILE-ADVICE
READ ORDER-FILE.
```

The same steps apply to `AFTER` advice.

When it comes to `AROUND` advice, effort is needed to handle the `PROCEED` statement. Consider this variation of the previous advice:

```
    USE AROUND READ
2   AND BIND VAR-NAME TO NAME OF FILE.
  FILE-ADVICE.
4   DISPLAY "Reading␣", VAR-NAME, ".".
    PROCEED.
6   DISPLAY "Done␣reading.".
```

First the original join point shadow is moved to a newly created section. For the example `READ` statement, this becomes:

```
READ16-JOIN-POINT SECTION.
JOIN POINT.
  READ ORDER-FILE.
```

Again, the names shown here are only indicative. Then, when cloning the body of the `AROUND` advice, all `PROCEED` statements are replaced by a `PERFORM` statement that executes the moved shadow:

```
AOP43-FILE-ADVICE SECTION.
FILE-ADVICE.
  DISPLAY "Reading␣", "ORDER-FILE", ".".
  PERFORM READ16-JOIN-POINT.
  DISPLAY "Done␣reading.".
```

Finally, the original paragraph for the shadow is re-implemented such that the cloned advice is invoked.

```
PERFORM AOP43-FILE-ADVICE.
```

Note that we have to prevent accidental fall-through of the control flow from the original sections into the added sections, which are therefore placed at the end of the procedure division after a special section that is also generated by the weaver:

```
DO-NOT-CROSS SECTION.
DO-NOT-CROSS-PARAGRAPH.
  GOBACK.
```

As for weaving an inter-program aspect, we can easily reduce this problem to the intra-program case by preparing each single program as follows: the program is composed with the environment and data divisions of the aspectual unit. This composition can be reverted if we fail to identify any relevant join point shadow for the program. This composition is subject to alpha conversion so that unintended name capture is avoided.

The weaving semantics, discussed so far, exhibits one scalability problem. Consider an aspect whose pointcut applies to many different shadows and whose advice code is of substantial size. In this (not too unrealistic) case, the pervasive cloning approach, as described above, will imply code explosion. If we want to reuse the advice code, as is, we can try to declare data items for the variables so that parameters are passed through global data items to the advice code. In the example, we would need this data declaration:

```
01 VAR-NAME PIC X(31).
```

The original advice would then be mapped onto:

```
AOP-FILE-ADVICE SECTION.
FILE-ADVICE.
  DISPLAY "Reading␣", VAR-NAME, ".".
```

Finally, the join point shadow is transformed as follows:

```
MOVE "ORDER-FILE" TO VAR-NAME
PERFORM AOP-FILE-ADVICE
READ ORDER-FILE.
```

This technique does not readily generalise for bindings that cannot be stored in Cobol data items, e.g., symbolic file names, even though the amount of cloning can still be reduced by reusing sub-paragraphs that do not refer to such problematic bindings. The technique is also challenged by `AROUND` advice as this would require parameterisation of procedure names. In fact, Cobol offers some related idioms. There exist vendor extensions for data items with `USAGE PROCEDURE-POINTER`, though this would make the weaver vendor-specific. There is also the possibility of using nested subprograms instead of procedures for join point shadows, for which the alphanumeric names can readily be stored. Even simpler, the weaver could just assign literal codes to all paragraphs, and then use a single 'monster switch' to map literal codes to actual `PERFORM` statements (see [Sne96a, p. 111] for an example of this).

3.9 Evaluation and critique

To give the reader a feel for the expressiveness of AOP+LMP in general, and of Cobble in particular, we will now present and discuss a few

example aspects. The first will be a logging/tracing aspect. The others will be closer to the real uses for Cobol; i.e. file-data manipulation.

3.9.1 Logging/tracing

The obvious example, is, of course, one of logging/tracing. Figure 3.1 shows a possible implementation for such a tracing aspect. Tracing is implemented to occur on two levels: tracing of statements (lines 7–16), and tracing of procedures (lines 18–27).

The first advice (`MY-STATEMENT-TRACING`, line 7) is set up to capture all statements (line 8). It then extracts information on the kind of statement which will be executed (`VERB`-selector on line 9), as well as the location in the source file (`LOCATION`-selector on line 10). This information is then used within the advice body to generate the trace (`DISPLAY` statements on lines 12 and 15).

The second advice (`MY-PROCEDURE-TRACING`, line 18) is put up in a similar fashion, only this time selecting all procedures (line 19). Also, we now extract the name of this procedure (`NAME`-selector on line 20) for use in the advice body.

Finally, the aspect has been defined as an *inter-aspect* (see section 3.6.2) by naming it as such (through the `ASPECT-ID` keyword on line 2). This will make it fit for easy reuse in a multitude of applications.

3.9.2 Handling unsafe access to file records

We will now devise an aspect that determines an error condition — even though Cobol's runtime system does not report any error whatsoever. We face the following assignment:

> *Any read access to a file's record (not to be confused with access to the file itself) is to be guarded by a test for the `FILE-STATUS` field to be equal to `ZERO` (meaning no unhandled error occurred previously).*

An implementation of this concern can be seen in figure 3.2. It features one advice (`MY-UNSAFEREAD-CONCERN` on line 9). As the scenario requires from us to shadow *sending data items*, we capture all statements and bind their sending data items (lines 10 and 11). Note

```
1  IDENTIFICATION DIVISION.
     ASPECT-ID. ASPECTS/TRACING.
3
   PROCEDURE DIVISION.
5    DECLARATIVES.

7      MY-STATEMENT-TRACING SECTION.
         USE AROUND ANY STATEMENT
9        AND BIND VAR-VERB TO VERB
         AND BIND VAR-LOC TO LOCATION.
11      MY-TRACING-ADVICE.
         DISPLAY "Before␣", VAR-VERB,
13               "␣statement␣at␣", VAR-LOC.
         PROCEED.
15       DISPLAY "After␣", VAR-VERB,
                 "␣statement␣at␣", VAR-LOC.
17
       MY-PROCEDURE-TRACING SECTION.
19       USE AROUND PROCEDURE
         AND BIND VAR-NAME TO NAME
21       AND BIND VAR-LOC TO LOCATION.
        MY-TRACING-ADVICE.
23       DISPLAY "Before␣procedure␣", VAR-NAME,
                 "␣at␣", VAR-LOC.
25       PROCEED.
         DISPLAY "After␣procedure", VAR-NAME,
27               "␣at␣", VAR-LOC.

29   END DECLARATIVES.
```

Figure 3.1: An aspect for tracing Cobol applications.

that if a statement has no sending data items, this selection will fail, and therefore the pointcut will not match.

The pointcut must be restricted to data items whose declaration is hosted in the file section. To this end, we use the `IS FILE DATA` condition (line 12; see also section 3.5.3). All that now remains is to select the relevant file-status field to be tested in the advice. This selection starts from the sending data item, and walks through the involved file to the

```
1  IDENTIFICATION DIVISION.
     ASPECT-ID. ASPECTS/UNSAFEREAD.
3  DATA DIVISION.
     WORKING-STORAGE SECTION.
5      COPY "BOOKS/PANIC.DD".
   PROCEDURE DIVISION.
7    DECLARATIVES.

9      MY-UNSAFEREAD-CONCERN SECTION.
         USE BEFORE ANY STATEMENT
11       AND BIND VAR-ITEM   TO SENDER
         AND      VAR-ITEM   IS FILE-DATA
13       AND BIND VAR-FILE   TO FILE OF VAR-ITEM
         AND BIND VAR-STATUS TO FILE-STATUS
15                              OF VAR-FILE
         AND BIND VAR-NAME   TO NAME OF VAR-FILE
17       AND BIND VAR-LOC    TO LOCATION.
        MY-UNSAFEREAD-ADVICE.
19       IF VAR-STATUS NOT = ZERO
           INITIALIZE PANIC-FIELD
21         MOVE VAR-NAME      TO PANIC-RESOURCE
           MOVE "UNSAFE␣READ" TO PANIC-CATEGORY
23         MOVE VAR-LOC       TO PANIC-LOCATION
           MOVE VAR-STATUS    TO PANIC-CODE
25         GO TO PANIC-STOP.

27   END DECLARATIVES.
```

Figure 3.2: An aspect for handling unsafe access to file records.

status field (lines 13–15). We also extract the name of the file involved (line 16), as well as the location of the (possibly offending) statement (line 17). Finally, the advice checks to make sure that the status field is zero. If it is not, then the context information is placed into some data fields, and control is handed over to a `PANIC-STOP` procedure.

3.9.3 Enforcing a file access policy

Let us consider an example for policy enforcement. In the previous section, we studied an aspect that caught unsafe read access to file records. This aspect exhibits several shortcomings. First, reading from the file record without a prior `READ` statement appears to be inappropriate, but the aspect will not spot this problem (*false negative*). Also, file-status fields can be shared among different files, in which case the aspect's insistence on a zero status prior to record access might be unnecessarily restrictive (*false positive*). What is needed is a waterproof file-access policy, where each file I/O statement must be executed in accordance to restrictions on the history of file access and record access in the given program. In figures 3.3 and 3.4, we cover part of this problem: we only care about the open/closed status of files. Other concerns can be handled in a similar way.

The aspect consists of three concerns — `OPEN` statements, `CLOSE` statements and other file I/O statements are treated differently. The open concern establishes that the file is not yet open, and marks it as open (lines 48–56). The close concern establishes that the file is still open, and marks it as closed (lines 58–66). The concern for other statements insists on the file being marked as open (lines 67–77). The implementation makes use of the `IDREF`-selector (see section 3.5.3), which delivers a unique reference to the given program entity. In the specific example, we need such unique references to distinguish the various files in the program at hand. We can maintain these `IDREF`s and the file-open status *per file* in a dynamic table —these will only be available in the Cobol 2008 standard though. If we know the maximum numbers of files beforehand (or set a limit), we can of course use a table of fixed size.

3.10 Conclusion

This chapter has shown how AOP and LMP can be integrated in an existing legacy language, in casu Cobol. We have extended the pointcut language of the aspectual extension in two ways. First we have given the pointcut structures backtracking and unification semantics. This can be done by overlaying the existing logic descriptions with this functionality. Secondly, we have added a mechanism for binding data from the pointcut matches to variables, and allowed these variables to act as

template parameters within the advice code. This made it possible to write generic advice which has access to the execution context. There is nothing in either of these extensions which would prevent them from being applied to other legacy languages. In fact, in section 5.1 we show this with another visit to ANSI-C.

```
1  IDENTIFICATION DIVISION.
     ASPECT-ID. ASPECTS/CHECKOPEN.
3
   DATA DIVISION.
5    WORKING-STORAGE SECTION.
       01 MY-IDREF         PIC 9(10).
7      01 CHECKOPEN-IDX    PIC 999.
       01 CHECKOPEN-MAX    PIC 999 VALUE ZERO.
9      01 CHECKOPEN-ENTRY OCCURS ANY TIMES
                    INDEXED BY CHECKOPEN-IDX.
11        05 CHECKOPEN-IDREF PIC 9(10).
          05 CHECKOPEN-STATE PIC 9.
13           88 CHECKOPEN-CLOSED VALUE 0.
             88 CHECKOPEN-OPEN   VALUE 1.
15
   PROCEDURE DIVISION.
17   DECLARATIVES.

19     MY-OPEN-CONCERN SECTION.
         USE BEFORE OPEN
21       AND BIND VAR-IDREF TO IDREF OF FILE.
       MY-OPEN-ADVICE.
23       MOVE VAR-IDREF TO MY-IDREF.
         PERFORM GET-CHECKOPEN-IDX.
25       IF CHECKOPEN-OPEN(VAR-IDREF)
           PERFORM CHECKOPEN-PANICS.
27       SET CHECKOPEN-OPEN(VAR-IDREF) TO TRUE.

29     MY-CLOSE-CONCERN SECTION.
         USE BEFORE CLOSE
31       BIND VAR-IDREF TO IDREF OF FILE.
       MY-CLOSE-ADVICE.
33       MOVE VAR-IDREF TO MY-IDREF.
         PERFORM GET-CHECKOPEN-IDX.
35       IF CHECKOPEN-CLOSED(MY-IDREF)
           PERFORM CHECKOPEN-PANICS.
37       SET CHECKOPEN-CLOSED(MY-IDREF) TO TRUE.
```

Figure 3.3: Policy checking for the status of files to be open (part 1).

```
38      MY-ACCESS-CONCERN SECTION.
          USE BEFORE
40        (READ OR REWRITE OR WRITE
                OR DELETE OR START)
42        AND BIND VAR-IDREF TO IDREF OF FILE.
         MY-ACCESS-ADVICE.
44        MOVE VAR-IDREF TO WORK-IDREF.
          PERFORM GET-CHECKOPEN-IDX.
46        IF CHECKOPEN-CLOSED(MY-IDREF)
            PERFORM CHECKOPEN-PANICS.
48    END DECLARATIVES.

50    GET-CHECKOPEN-IDX.
      *> Find or allocate entry for VAR-IDREF
52    *> in dynamic table.

54    CHECKOPEN-PANICS.
      *> Print error message and stop execution.
```

Figure 3.4: Policy checking for the status of files to be open (part 2).

Chapter 4

The transformation framework

If you want to make an apple pie from scratch,
you must first create the universe.

CARL SAGAN

UP until now we have considered how to bring Aspect-oriented Programming and Logic Meta-programming into legacy languages, only from the point of view of those languages. In doing so we have described in what ways the original languages should be modified, and in what ways the advice code should be woven. It is now time to ask how this is achieved in practice. To this end this chapter now presents the transformation framework which is used to do this.

4.1 XML representations of source code

The framework which will be expanded on in the following sections, is one which was custom built in the context of the ARRIBA project (see section 1.1.1): *Yerna Lindale*.[1] It is governed by one major design decision which was of importance to the project: it operates at the level of source code and uses XML for the internal representation of that code.

[1] *Yerna Lindale:* Quenya (High Elvish) for *'old music'* — this for fans of J. R. R. Tolkien.
Download: `http://users.ugent.be/~kdschutt/yernalindale/`

Figure 4.1: The weaving process.

Put in another way: the abstract syntax tree (AST) for the source code is made accessible in XML form. Weaving then takes place through transformations of this XML structure. Figure 4.1 captures this idea.

What about alternatives? Weaving compiled code would imply commitment to a specific vendor including dialect and object format. Language-independent load-time weavers [LC03] (for .NET or otherwise) would be challenged by the distance between bytecode and the pointcut descriptions. Language-independent weavers at the source-code level, such as SourceWeave.NET [JC04], would require specific language front-ends that appeal to the underlying source-code models (i.e., CodeDOM for SourceWeave.NET).

The use of XML for the intermediate representation of source code is not uncommon; it is practised for 'normal' languages (such as Java [Bad00] and C [ZK01]) and also for languages with a weaving semantics. [GBNT01, SPS02] Note that the focus on XML does not mean that it is the only viable option for these things. Rather, any format for representing trees will do. The important advantage of using XML is that standard APIs and tools for XML processing can be readily used. This buys us great flexibility and opportunities for experimentation with different technologies (this indeed was the major reason it was chosen for use in ARRIBA).

4.2 Component integration framework

The actual weavers are implemented around the XML structure using a number of different languages:

- A Prolog component is used for matching pointcuts to join-point shadows. We use a specific Prolog implementation, TuProlog[2], which is fully inter-operable with Java. (For the record, there is no proper obstacle to reconstructing the component for matching

[2] `http://lia.deis.unibo.it/research/tuprolog/`

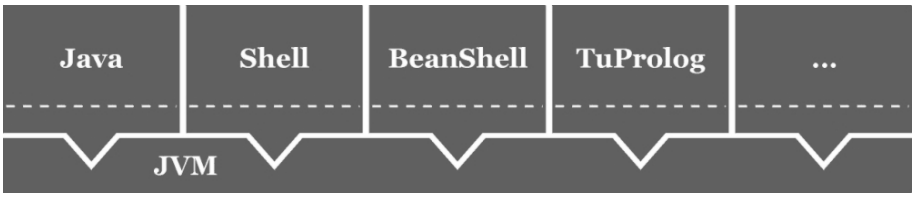


Figure 4.2: Component integration framework.

pointcuts in plain Java.)

- The actual weavers are implemented in Java/BeanShell[3] using XML APIs for DOM. DOM is a standardized representation of the XML trees, together with a standardized interface for accessing and modifying it.

- The source code parsers and, hence, the translators to XML, are implemented in C (using BTYACC for the Cobol parser) and Java (ANTLR, for the C parser).

Integrating these various components, each of which may have been written in a language of its own, is in fact quite straightforward. The component integration framework, *Lillambi*[4], leverages the fact that all components are written in some language that operates on the Java Virtual Machine, or can be accessed from it (see figure 4.2). The main advantage is that all data being processed is in the form of Java objects. Hence, communication between components requires no exotic technologies.

Structurally, each component gets wrapped in an "Agent" interface:

```
public interface Agent {
  public Object [] act (Object [] args)
    throws AgentException;
}
```

As can be seen, each agent has one entry point: the `act` method. It accepts an array of objects, which can be used to pass any number of arguments to the agent. The agent will respond by returning a similar array.

[3] `http://www.beanshell.org/`

[4] *Lillambi:* Quenya (High Elvish) for '*many tongues*' — this, again, for fans of J. R. R. Tolkien. Download: `http://allserv.ugent.be/~kdschutt/lillambi/`

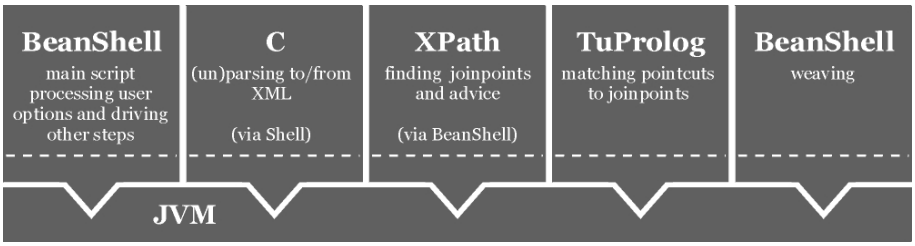


Figure 4.3: Lillambi setup for Cobble.

As an aside, we invoke agents written in C using system calls. The use of Java Native Interfaces (JNI) would, of course, provide a more tightly integrated solution. When it becomes necessary we can easily do this switch.

The idea is that each agent performs one specific kind of functionality. Thus, we have one agent doing the parse to XML, one agent doing the join point shadow extraction, one doing the advice extraction, one doing the matching between join points and advice, one doing the actual weaving, one doing the transformation back to source code, and, finally, one which orchestrates all the others. Figure 4.3 shows this setup as it is for Cobble.

Agents are associated with distinct names, and can be retrieved through the main registry class:

```
public class Lillambi {
  public static Agent find (String agentID)
    throws AgentNotFound, AgentNotLoaded {...}
}
```

The result is a simple framework in which different concerns can be tackled using the most appropriate technique, while still being able to integrate each component into a functional whole.

4.3 Front-end of the transformation framework: source to XML

According to Gray and Roychoudhury in [GR04], the first obstacle for AOP support lies in getting a handle on a front-end for the relevant legacy language (i.e. the parsers). In the context of the Yerna Lindale framework this means parsing source code into an XML format. This

```
display :
  "DISPLAY"
  (what_to_display
    where_to_display? display_option*)+
  ("ON"? "EXCEPTION" statement+)?
  ("NOT" "ON"? "EXCEPTION" statement+)?
  "END-DISPLAY"?;

what_to_display : identifier | literal;
```

Figure 4.4: LLL Grammar fragment for the DISPLAY statement

is indeed a major challenge in the case of Cobol, as has been argued elsewhere. [LV01].

4.3.1 The front-end for Cobble

Cobble's front-end is based on a consolidated VS Cobol II grammar[5], which was recovered from IBM's language reference in a project by the Vrije Universiteit Amsterdam. [LV01] Recovery of this grammar was based on the Grammar Recovery Kit[6] (GRK), which uses scripts for transforming grammars, and the Grammar Deployment Kit[7] (GDK), which can transform grammars into usable parsers. The parser for Cobble is based on a grammar (recovered in a similar way) for AcuCobolGT, a choice which was made based on the availability of the ACUCobolGT compilers and tools.

The GRK and the GDK interoperate via an EBNF-like grammar formalism, *LLL*, from which the GDK can generate parsers based on different technologies. (Fig. 4.4 shows an LLL excerpt of one of our Cobol grammars.) This in turn is transformed by the GDK into a BTYACC[8] parser. The choice for BTYACC is a pragmatic one: the recovered Cobol grammars are still ambiguous (no context-free grammars for Cobol exist), and BTYACC can cope with this ambiguousness through the use of a backtracking mechanism.

The grammars were subsequently extended to offer concrete syn-

[5] http://www.cs.vu.nl/grammars/vs-cobol-ii/
[6] http://www.cs.vu.nl/grammarware/grk/
[7] http://gdk.sourceforge.net/
[8] http://www.siber.com/btyacc/

tax for all AOP language elements (as opposed to AOP technologies that use XML for join-point description, such as JBoss AOP and AspectWerkz). This was achieved by making use of the GRK's transformation scripts. Figure 4.5 shows part of this script, namely the redefinition of the declaratives section, and the (start of) the addition of the aspect grammar.

One extra complication lay with Cobol's pre-processor directives; most especially regarding its `COPY` statement, which acts in a similar way to `#include` directives in C. As the parser we generate cannot deal with these directives, and as the grammars can not reasonably be extended to cover all possible positions for them, this requires us to pre-process Cobol sources before feeding them to the parser.

4.3.2 The front-end for Wicca

Development of the front-end for Wicca was performed along similar lines as the previous section. A first grammar was found online[9], and was subsequently transformed to comply to the GDK's format. As the grammar for C is significantly more manageable than that for Cobol, this time we opted to modify this grammar in-situ in order to add AOP constructs. Fig. 4.6 shows part of the result of this effort. From this, we again generated a BTYACC-based parser. However, as we could have a context-free C grammar, there really was no need for BTYACC's backtracking solution. Instead, we preferred a more strict parser which could detect errors in input more quickly and with more authority. For this reason, Aspicere (Wicca's follow-up, see section 5.1) discarded our first parser in favour for one based entirely on ANTLR.[10]

4.3.3 XML-ification of the parse trees

In all cases, the parser is extended by a transformer that maps the abstract syntax trees to XML (step 1 in Fig. 4.1). The XML representation encodes all details of layout and comments, which is less important for AOP, since a user is not supposed to study the woven code. Figure 4.7 illustrates this XML representation.

There are known scalability problems here, which require extra effort for compact XML representations or the use of tool-to-tool XML

[9] `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`
[10] `http://www.antlr.org/grammar/cgram`

```
% Rewriting the declaratives...
Reject declaratives;

% Extend the original declaratives...
Add
  declaratives =
    "DECLARATIVES" "."
    (declarative-section | advice)+
    "END" "DECLARATIVES" "."
;

% Make sure the old definition still works...
Add
  declarative-section =
    section-name "SECTION" segment-no? "."
    declarative-sentence
    paragraph*
;

% Definition for advice...
Add
  advice =
    section-name "SECTION" "."
    advice-quantification
    paragraph*
;

% The AOP version of the use statement...
Add
  advice-quantification =
    "USE" advice-when pointcut "."
;

Add
  advice-when = "AFTER" | "BEFORE" | "AROUND"
;
```

Figure 4.5: Excerpt of the transformation script for adding AOP constructs to a Cobol grammar.

```
wicca_application
  : ( function_definition
    | declaration
    | advice_definition )+
  ;

advice_definition
  : return_type
    "advice" pointcut compound_statement
  ;

pointcut
  : and_ppc
  | and_ppc righthand_or_ppc+
  ;
righthand_or_ppc : "||" and_ppc ;

and_ppc
  : unary_ppc
  | unary_ppc righthand_and_ppc+
  ;
righthand_and_ppc : "&&" unary_ppc ;

unary_ppc
  : "!" primitive_ppc
  | primitive_ppc
  ;

primitive_ppc
  : on_ppc
  | "(" pointcut ")"
  ;

on_ppc
  : "on" "(" ( identifier | regexp ) ")"
  ;
```

Figure 4.6: LLL Grammar fragment for Wicca

```
<display>
 <string>DISPLAY</string> <!--   -->
 <what_to_display>
  <literal>
    <string>&quot;HELLO WORLD!&quot;</string>
  </literal>
 </what_to_display>
</display>
```

Figure 4.7: XML element for `DISPLAY "HELLO WORLD!"`

APIs without intermediate textual XML content. [Sim00] In our prototype, we currently neglect these issues. We can report that the ratio *concrete syntax to XML format* (both in text representation) is typically 10, which is still quite tractable.

As can be seen in figure 4.7, the XML format is expressed in terms of the grammar (see figure 4.4 for the corresponding grammar fragment). As a consequence, all functionality that operates on the XML representation does not resist grammar changes. Also, we cannot directly serve multiple language-dialects (i.e. Cobol). Existing work on language-independent source-level weavers [JC04], and, more generally, on language implementation should be of use in this context.

The transformation to XML is complemented with an unparser (step 3 in Fig. 4.1) which maps XML data back to concrete (Cobol, C,...) syntax. The various XML element types correspond to the nonterminals in the grammars.

While it may seem that Yerna Lindale relies heavily on GDK-generated parsers this is not really the case. As long as we have a parser which can generate XML dumps of its ASTs our framework will be able to handle things.

4.4 Back-end: XML-based source code querying

The second obstacle for AOP support, put forward by Gray and Roychoudhury [GR04], concerns the weaving framework. At the very least, we require a basic transformation framework, which allows us to express and apply the weaving semantics (by which we mean the elimination of Aspect-oriented constructs in terms of transformations). The

```
function_definition_name(Def, Name) :-
  child1_named(Def, 'declarator', Decl),
  child1_named(Decl, 'direct_declarator', DD1),
  child1_named(DD1, 'direct_declarator', DD2),
  child1_named(DD2, 'identifier', Identifier),
  child1_named(Identifier, 'string', String),
  child1(String, Text),
  node_value(Text, Name),
  !.

function_call_name(Call, Name) :-
  child1_named(Call, 'postfix_expression', PF),
  child1_named(PF, 'primary_expression', Prim),
  child1_named(Prim, 'identifier', Identifier),
  child1_named(Identifier, 'string', String),
  child1(String, Text),
  node_value(Text, Name),
  !.
```

Figure 4.8: Example of handwritten XML queries.

weavers in Yerna Lindale process XML via DOM, and generate new XML (step 2 in Fig. 4.1). This section will focus on the task of querying the structure.

4.4.1 Querying XML using PAL

The first incarnation of the weaving framework was driven entirely from Prolog. This meant that we had to extract information out of the XML through Prolog predicates. Writing this navigation and extraction code by hand proved labour intensive and error prone. Figure 4.8 shows an example of this. It features two queries, the first for finding a name of a procedure definition (that is, the name of the defined procedure), the second for finding the name of a procedure call (that is, the name of the procedure being called).

To make XML queries and the extraction of data from XML easier, we developed the *PAL* format. A pal-file is in essence nothing more than an lll grammar which has been annotated with Prolog code (hence: PAL, or *Prolog-Annotated LLL*). There is a tool, "pal_to_pl", which takes

```
#> wicca.lll
#@ advice
#- cutdown
#: wicca_application

wicca_application {= Advice}
  : <Collected:
      (function_definition
      | declaration
      | advice_definition)+
    >
    { no_empties(Collected, Advice) }
  ;

advice_definition {= [PC, RVal, Body, This]}
  : <RVal: declaration_specifiers?>
    "advice" <PC: pointcut>
    <Body: compound_statement>
  ;

pointcut {= Collected }
  : <Collected: and_ppc>
  | <Left: and_ppc> <Right: righthand_or_ppc+>
    { Collected = [or, Left, Right] }
  ;

righthand_or_ppc {= PPC}
  : "||" <PPC: and_ppc> ;

% and_ppc, unary_ppc and primitive_ppc...

on_ppc  {= [on, Pointcut] }
  : "on" "("
    <Pointcut: ( identifier | regexp)>
    ")"
  ;
```

Figure 4.9: Fragment of the PAL description for extracting advices.

pal-files, and outputs the equivalent Prolog navigation and extraction code. Put differently, PAL applies a technique of attribute grammars (similar to Yacc) to facilitate extraction of data.

An example pal-file (used for Wicca) is shown in figure 4.9. Lines 6 through 36 make up the annotated grammar (compare this to the original grammar in figure 4.6). These are preceded by some directives.

Line 1 names the reference grammar which the pal-file is tied to. This enables the tool to find any grammar rules which have not been repeated in the pal-file, thus allowing the annotations-writer to focus his/her work on the task at hand.

As we want to extract different kinds of information (e.g. join point shadows, advice definitions, function definitions,...) from the same structure using the same grammar, we need a way to disambiguate these different tasks. This is done by naming this task in an annotation, which is what happens on line 2.

Line 3 lets the tool know that the generated prolog code should only do matching on children if the value which that child returns is bound to a name (as `Pointcut` is on line 34). In addition, cutdown will not generate predicates for grammar rules which are not active (i.e. whose return value is never bound). This helps minimize the size of the generated prolog code, as well as the time spent extracting information. Of course, to know which grammar rules are active we need to know which rules are used by the programmer in the first place. This is what the starter directive on line 4 is for.

Note that not doing a cutdown has its uses: it will match the *full* structure of the DOM tree against the grammar. This allows one to verify whether or not the DOM tree is correct with respect to the associated grammar (similar to a DTD verification), which can prevent otherwise mysterious errors.

The directives are then followed by the annotated grammar. To show what happens here, let us start by taking a closer look at the definition for `righthand_or_ppc` on line 27. Here it is again:

```
righthand_or_ppc {= PPC}
  : "||" <PPC: and_ppc> ;
```

The first line declares that the value returned when matching this structure is the value associated with `PPC`. The second line shows that this variable is tied to `and_ppc`. The net effect is this: when matching the structure for `righthand_or_ppc`, the result obtained from matching

`and_ppc` is tied to the `PPC` variable, which is then returned as the result of the match.

The basic values which can be passed along are either references to XML nodes which have been matched (this is the default return value for grammar rules which were not annotated), or Prolog structures. For an example of the latter, see the definition of `on_ppc` (line 32), which returns:

```
[on, Pointcut]
```

This is a basic Prolog list. The first element is simply the symbol 'on' (used here as a tag). The second element captures the pattern from the binding named `Pointcut`.

Similarly we can see that `pointcut` (line 21) builds a list representing the disjunctions in a pointcut. This is further composed in `advice_definition` (line 15) in a structure representing the full advice. E.g., given a source advice as in:

```
int advice on (.*) && ! on (printf) {
  /* body */
}
```

This would have `advice_definition` return (with `R`, `B` and `A` as placeholders for references to XML nodes):

```
[[and,[on, '.*'],[not,[on,'printf']]], R, B, A]
```

Figure 4.10 shows a fragment[11] of what gets generated behind the scenes to make all this work. It features the Prolog translations of matching the `wicca_application` grammar rule (lines 1–9, and subphrase on lines 28–37), as well as the `advice_definition` grammar rule (lines 11–26).

Given all this, matching the DOM tree and extracting information becomes straightforward. Starting from a reference to the root node of the DOM tree, we can now find all advices, simply by writing:

```
match_term(
  advice,      %> name of the set of annotations
  wicca_application,    %> grammar rule to match
  Root,           %> node which is to be matched
  Advices      %> the structure which gets built
)
```

[11] The fragment has been edited to make it fit into the available space.

```
% Matching 'wicca_application'...
match_term(advice, wicca_application,
           This, Advice) :-
  xml_name(This, 'wicca_application'),
  pal_first_element(This, PAL_1ST),
  match_term(advice, phrase1,
             plus, PAL_1ST, none, C0),
  Collected = C0,
  no_empties(Collected, Advice), !.

% Matching 'advice_definition'...
match_term(advice, advice_definition,
           This, [PC, RVal, Body, This]) :-
  xml_name(This, 'advice_definition'),
  pal_first_element(This, PAL_1ST),
  match_term(advice, declaration_specifiers,
             opt, PAL_1ST, PAL_NXT_0, C0),
  RVal = C0,
  scan_literal(advice, 'advice',
               once, PAL_NXT_0, PAL_NXT_1),
  match_term(advice, pointcut,
             once, PAL_NXT_1, PAL_NXT_2, C2),
  PC = C2,
  match_term(advice, compound_statement,
             once, PAL_NXT_2, none, C3),
  Body = C3, !.

% Matching phrases nested in rules...
match_term(advice, phrase1,
           PAL_1ST, PAL_COMPOSITE) :-
  ( match_term(advice, function_definition,
               once, PAL_1ST, PAL_NXT, C0)
  ; match_term(advice, declaration,
               once, PAL_1ST, PAL_NXT, C0)
  ; match_term(advice, advice_definition,
               once, PAL_1ST, PAL_NXT, C0)
  ), PAL_COMPOSITE = C0, !.
```

Figure 4.10: Excerpt of Prolog code generated from figure 4.9.

If the match can be made, then Advices will be unified with a list of all advices in the DOM tree, each represented through a structure as the one above.

4.4.2 Querying XML using XPath

When working with XML structures we can make use of the technologies and tools surrounding them. One of these is XPath. Using XPath, which can be used to locate pointcuts, advice and potential join-point shadows in a DOM tree. For instance, to find all Cobol paragraphs in a DOM tree, the following query suffices:

```
//paragraph
```

This selects all XML nodes named "paragraph", anywhere in the tree ("//" means: at this level or any sublevel in the tree). The result of such queries will always be a set of XML nodes. To move from these to a structured representation of, for instance, pointcut descriptions (as was done in the previous subsection), the user still has to extract this data manually from those nodes (e.g. using `getNodeValue` calls in Java). Hence, a tool such as PAL is still of value. In fact, this is almost how Cobble and Wicca are implemented. But rather than using a full-fledged XPath engine, it uses the DOM API to perform the above query:

```
NodeList list
  = document.getElementsByTagName("paragraph");
```

The result set is subsequently iterated over, and each item gets matched, and its data extracted, by using the PAL rules.

4.5 Back-end: XML-based source code weaving

In Yerna Lindale, weaving source code entails the transformation of DOM structures. This means that a lot of code gets written which generates such structures. To do this all by hand is tedious and error-prone. We therefore opted for bringing in a tool to aid the programmer. We defined a special syntax which allows the programmer to write down new DOM structures quickly and concisely, *within* the hosting programming language[12] (sofar: Prolog and Java are supported). An example of this

[12] XJ is a more advanced example of the same principle, but one which was not available at the time of the construction of our framework. `http://www.alphaworks.ibm.com/tech/xj`

```
<< @Document | function_definition @Func :
     @DeclSpec
     declarator (
       direct_declarator (
         direct_declarator (
           identifier (
             @FunctionNameNode
           )
         )
         "("
         @ParTypeList
         ")"
       )
     )
     @AdviceBody
>>
```

Figure 4.11: Example of a DOM annotation.

can be seen in figure 4.11.

This notation is called a *DOM annotation*. The start and end of such a structure are indicated by double "fish-grates" (lines 1 and 16). The first line identifies the root node which gets built. In this case, we are constructing a `function_definition`. This root node will be bound to the `Func` variable, which is a variable defined in the hosting language. In order to be able to create anything in DOM, we need access to the document. This is the very first parameter, `Document`, which can be seen in the example.

After these preliminaries, following the colon on line 1, comes a definition of the structure as it should be built. The syntax here distinguishes between two types: node-names, and actual nodes (indicated with a leading "@"). The first declare what the name of a new node must be. The latter are references to existing nodes, and are used to place such nodes in the new structure.

Parent-child relationships are indicated through parentheses. That is, if a node has children, then these children are defined enclosed in braces and immediately follow the parent node:

```
parent ( child_node other_child_node )
```

The example adds extra visibility to this by also indenting child nodes. Barring these braces, a sibling relationship is assumed:

```
node sibling_node
```

There is one extra piece of syntactic sugar. As can be seen in the example on lines 10 and 12, string literals may also be included. These will be transformed into textual data for the DOM structure.

The obvious alternative to these DOM annotations would be the use of XSLT. This however has the disadvantage of being less closely integrated with the language, which makes it therefore harder to embed into the transformation algorithms. It would also entail duplication of a lot of data, as XSLT does not modify the original DOM structure, but instead generates a new one.

4.6 Supported platforms

As the core of Yerna Lindale relies on Java technology, the transformation framework is platform-independent to a large degree. Whether or not it can be made to run on a certain system depends mostly on whether the tools Yerna Lindale relies on (the Grammar Deployment Kit, BTYACC, Bash, etc.) are available, or can be made available, on that system.

Sofar, the framework has been tested on several distributions of Linux (among them Redhat, Mandrake, Ubuntu), and there is no technical reason why it shouldn't work on others. We have also got the framework up and running on Mac OS X machines. Support for Windows has not been tested yet, but given environments such as Cygwin this too should work.

The Cobol parsers were tested against a large portfolio of industrial Fujitsu-Siemens 2000 Cobol code (959 KLOC in 35MB of code, 21.8MB of which is dedicated to copy files), as well as against the examples provided by AcuCobol in their distribution. The C parser has been tested against several applications, one of which was an industrial legacy system (453 KLOC).

4.7 Applying the framework to other languages

We have seen that the transformation framework is built around XML representations of the abstract syntax trees (AST) of source code. The framework itself can then be built out of different components, using different technologies, which may then be integrated through a special interface (see section 4.2).

In order to apply this framework to another language (i.e. a language which is not already supported), in essence al that's needed is a source code parser which can dump the ASTs to XML. The format for this XML tree is very straightforward:

- XML nodes represent the grammar rules as which its children have been classified. I.e. if a token is name which is part of a function definition, then that token will be a child of a "name" node, which in turn is a child of a "function-definition" node.

- Whitespace is (optionally) encoded in XML comments, with newlines in whitespace represented as "@n".

This parser can then be wrapped as an agent (see section 4.2), after which it is accessible from other agents/components.

The addition of AOP and LMP requires some extra effort, of course. First one must define the join points of interest, and based on those construct a pointcut language and its semantics. (Chapters 2 and 3 did exactly this for ANSI-C and Cobol). From this one must distill a grammar, and turn it into a parser which generates XML (as above). Once this step is completed it can be integrated into the framework. From then on the tools and techniques presented in this chapter can be applied to query and transform (and thus weave) source code in the new language.

Reuse of existing weavers for weaving code in new languages is only possible if the XML format for source code of the new language matches that of another language. As this format is strongly tied to the grammars of those languages, this is unlikely to be the case. This might be solved by defining a common abstraction between languages, and have the weavers work on this abstraction, but the question is how arbitrary transformations of the abstracted representations should map back to equivalent transformations of the concrete representations. A lot of important details which are required to generate correct source

code have to be left out at the abstracted level, and as such are lost when performing transformations. It is unclear as to how this detail should re-emerge.

4.8 Conclusion

This chapter presented a transformation framework which has successfully been applied to enable AOP and LMP in two very different programming languages: C and Cobol. As such it has proven its usability across a wider range of languages. The framework is based around XML DOM-tree representations of the abstract syntax trees of source code, and the transformation thereof. The advantage of this approach is that it may be integrated with a lot of different tools which are capable of working with XML, something which can be seen in its application to C and Cobol. A framework based on XML is, of course, not a necessary requirement for enabling AOP and LMP in legacy languages. Any tool which is capable of transforming source code may be used. The added advantage of the framework is that it is flexible enough to work on many different kinds of languages.

Chapter 5

Validation of AOP and LMP for legacy software

Knowledge is of no value unless you put it into practice.

HEBER J.GRANT

WE have seen how Aspect-oriented Programming and Logic Meta-programming can be embedded in legacy languages, both conceptually and practically. In this chapter we now delve further into what these tools can do for us. We show how AOP and LMP can be fully enabled in ANSI-C. We present how the transformation framework was used for source code visualisations. We also discuss the application of AOP and LMP to several, very different problems with legacy software. We present the problems themselves, and explore how we might code our aspects, using the languages discussed in earlier chapters.

5.1 Aspicere: aspectual extension for ANSI-C

Aspicere[1] is a spin-off[2] of the work on aspectual extensions for C (chapter 2), integrating the meta-programming approach to pointcuts seen in the work on aspectual extensions for Cobol (chapter 3). As such it acts

[1] Download: `http://users.ugent.be/~badams/aspicere/`

[2] Work done by Bram Adams, GH-SEL, INTEC, UGent.

```
any advice on (.*) && ! on (printf) {
  any r = 0;
  printf ("before␣%s\n", this_joinpoint()->name);
  r = proceed ();
  printf ("after␣%s\n", this_joinpoint()->name);
  return r;
}
```

Figure 5.1: Generic tracing advice, as we would like it.

```
Type around tracing (Type) on (Jp):
  call (Jp, ".*") && ! call (Jp, "printf")
  && type (Jp, Type)
  && ! str_matches ("void", Type)
{
  Type r = 0;
  printf ("before␣%s\n", Jp->name);
  r = proceed ();
  printf ("after␣%s\n", Jp->name);
  return r;
}
```

Figure 5.2: Generic tracing advice, as Aspicere allows it.

as another validation that the AOP+LMP concept can be embedded in legacy languages.

5.1.1 Generic tracing advice

As was discussed in section 2.9, the first incarnation of aspects in C, Wicca, had some important limitations. A generic tracing advice, for instance, was not attainable. Consider again the code in figure 5.1. The `any` type shown in this example was wishful thinking. C does not know it, and Wicca provided no auto-casting feature to support it. However, through the use of LMP in the pointcut-mechanism, Aspicere will allow us to write generic advice, even without the need for an auto-casting feature. Figure 5.2 shows how we can achieve this.

As can be seen, the fictitious `any` type has been replaced with a

kind of *template parameter*, `Type`. This variable is declared as a template parameter in the advice parameter list on line 1. Its value is extracted from within the pointcut (line 3), in a similar way as bindings in Cobble (section 3.5.3). The result is that whenever this advice is matched to a join point, it will get bound to the correct type.

It is important to note the type restriction on line 4. This is needed because the advice, as it is written here, can not be instantiated for `void` types: the variable definition on line 6 would not make sense. Hence, we need a second advice for `void` procedures. This limitation is caused by how C treats the `void` type, not by how Aspicere integrates its LMP mechanism.

Apart from the extended pointcut mechanism, and its associated propagation of bindings, Aspicere is very similar to Wicca. There is a small difference in the declaration of advices, where Aspicere opts for named advices. There is also the choice to explicitly declare advice parameters for use within the advice (i.e. only the values for these parameters are made accessible from within the advice). This small cosmetic difference, however, hides a much greater flexibility and expressiveness. The following subsection will try to showcase this.

5.1.2 Generic parameter checking (ASML)

The following discussion is based on work done by Bruntink, Van Deursen and Tourwé, in the context of ASML, the world market leader in lithography systems. We will not repeat all details here, apart from those relevant to our discussion. For more details, we refer the reader to [BvDT04].

The context, put briefly, is this: the code produced by ASML has, among other things, three programming conventions tangled into it: error handling, parameter tracing and parameter checking. As the language they use, C, does not provide support for crosscutting concerns, ASML resorts to an idiomatic approach for implementing these concerns. I.e., programmers have to follow strict guidelines, and add these concerns manually.

Code for the parameter checking concern, which is what we will focus on for the remainder of this section, looks as follows:

```
1 if (queue == (CC_queue *) NULL) {
    r = CC_PARAMETER_ERR;
3   CC_LOG (r, 0,
```

```
      ("%s: Input parameter error (NULL)",
5                                  func_name));
  }
```

This checks whether `queue`, an *input* parameter, has been properly initialized. If not, an error code is set, and the error is logged.

Aside from *input* parameters, there are also *output* parameters, which are checked in the exact same way. The code differs only in the string that gets logged.

Additionally, there is a special class of *output* parameters, *output pointer* parameters, which are used as pointers to locations where the output should be placed. A check needs to be made that these locations contain no other data, or memory leaks may occur. The code for this looks like:

```
  if (*item_data != (void *) NULL) {
2   r = CC_PARAMETER_ERR;
    CC_LOG (r, 0,
4     ("%s: may already have data (!NULL).",
                                    func_name));
6 }
```

This is the same code as with other parameters, apart from the need of an extra dereference (line 1).

Bruntink et al. then go on to show how this concern may be encoded using an AOP approach, using AspectC (see 2.2.1). They are however hampered by the inability to generically handle parameters and their types, which forces them to write one advice per possible variation of the parameter list. E.g.:

```
  pointcut pc50 (void ** x) :
2   args (x) &&
    (execution (* CC_queue_data (..)) ||
4    execution (* CC_ReadMsg (..)) ||
     execution (* queue_extract (..)) ||
6    execution (* CC_iterator_peek (..)));

8 before (void ** x) : pc50 (x) {
    if (*x != (void *) NULL) {
10    r = CC_PARAMETER_ERR;
      CC_LOG (r, 0,
12      ("%s: may already have data (!NULL).",
```

```
                                                func_name));
14    }
  }
```

This captures execution of four different procedures (lines 3–6), each of which expects one argument of type `int` (line 2). Similar advices must be written for all possible variations of parameter lists. For the component they experimented on, this leaves them with 39 advices to cover the checking of all *output pointer* parameters. Again, this is due to the inability of quantifying over the structure of parameter lists and the types thereof. (Note that this is a problem for ANSI-C. Other languages may not necessarily suffer from this.)

Aspicere, with its LMP approach embedded into it, can do better. By quantifying the pointcut over all parameters at a join point, along with extracting its type information, we can cut the number further down to *two*. Here is the main one:

```
1 ProcType around leak_check
    (ProcType, ProcName, DerefType, Arg)
3   on (Jp):
      call (Jp, ProcName, Arguments)
5     && member (Arg, Arguments)
      && is_outputpointer (Arg)
7     && type (Arg, ArgType)
      && dereferenced (ArgType, DerefType)
9     && type (Jp, ProcType)
      && ! str_matches ("void", ProcType)
11 {
    if (* Arg != (DerefType) NULL) {
13    r = CC_PARAMETER_ERR;
      CC_LOG (r, 0,
15      ("%s:␣may␣already␣have␣data␣(!NULL).",
                                      ProcName));
17  }
    return proceed ();
19 }
```

The key to this advice can be found on lines 4 and 5. Line 4 captures all arguments a call may have, in `Arguments`. Line 5 then selects one of these, through the Prolog `member/2` predicate, and binds it to `Arg`. Note that `member/2` is ambiguous, and may have multiple solutions. According to the matching semantics as described in section 3.7.2, the

pointcut will match all possibilities in turn, and the advice will be instantiated for each of these solutions.

Line 6 asserts that the argument is an *output pointer*. This assertion can be done in one of two ways:

- Best-case scenario: checking whether a parameter is an *output pointer* parameter can be done by analysing (the structure of) the code. We could then encode this analysis into a rule, which would then be invoked during the matching of the join point.
- Worst-case scenario: the check relies on domain information, which is locked up in the heads of the developers. This means that we have to manually specify[3] one fact for each *output pointer* parameter which exists. These facts can then be used during the join point matching. The added advantage is that the previously implicit domain knowledge has now been made explicit, and is directly available for practical use by anyone.

Note that the test is made on the original argument and not on its type; ie. on `Arg` rather than on `ArgType`. This is needed because the analysis which may have to take place will require knowledge about the original argument; the type alone will most likely be insufficient.

The remainder of the pointcut is there to capture the right context information (mostly type information) for use in the advice. E.g. the type cast on line 12, and the procedure name on line 16.

As we said, two advices are necessary. The second one is needed to handle void procedures, for reasons discussed in the previous subsection.

The above advice provides only a basic example of what can be achieved using the technique of LMP embedded in AOP. In [AT05], Adams and Tourwé delve deeper into the parameter checking concern, making use of Aspicere to, among other things, prevent spurious parameter checks.

5.1.3 Conclusion

We have shown that AOP and LMP can be fully integrated into ANSI-C, and that, in doing so, we are able to write down truly generic aspects. We have argued this point by discussing the features of Aspicere

[3] The use of *annotations* might be of value here.

against a real-life industrial case, and have demonstrated how LMP helped make up for the lack of reflection in ANSI-C.

5.2 Runya: source code visualisation

As we have seen in section 1.1.1, a good understanding of the legacy applications is crucial to their further evolution. Yet, as such knowledge is missing, and as documentation is most often (at best) not up to date, reverse engineering is necessary. Runya[4] is an experiment[5] in the visualisation of legacy source code, and this in order to help reverse engineering efforts.

Inspiration for this project came from Rusty Russell's "Linux Kernel Graphing Project" (LKGP[6]), which processes the Linux kernel code, and outputs a visualisation thereof, down to the control flow within the different procedures. The nice thing about such a visualisation is that it makes it much easier for a human to grasp the scope and structure of the underlying source code. It provides a useful birds-eye view in one image. When it comes to grasping the structure of legacy source code, which often has but very little up-to-date documentation, extracting such a global picture from the sources could provide a useful starting point. Hence this experiment.

5.2.1 Approach

The tool developed for the LKGP project, however, was not directly useful to us. For one, it was targeted at the Linux kernel code, making use of its style-guide. For another, its visualisation lacked flexibility for displaying other projects. So we set off building a new tool, starting from the XML representations of source code which we could already generate with ease (see chapter 4).

The approach used for the tool is very simple. We start from the XML structures, and query these to extract the necessary information (control flow, procedure size, etc.). This is then put into a new (scene-graph) structure, which can be readily displayed using Garp, a custom derivative of the "G graphics library[7]".

[4] *Runya:* Quenya (High Elvish) for '*footprint*' — this for fans of J. R. R. Tolkien.
[5] Prototype implemented by Stijn Van Wonterghem, GH-SEL, INTEC, UGent.
[6] http://fcgp.sourceforge.net/lgp/index.html
[7] http://geosoft.no/graphics/

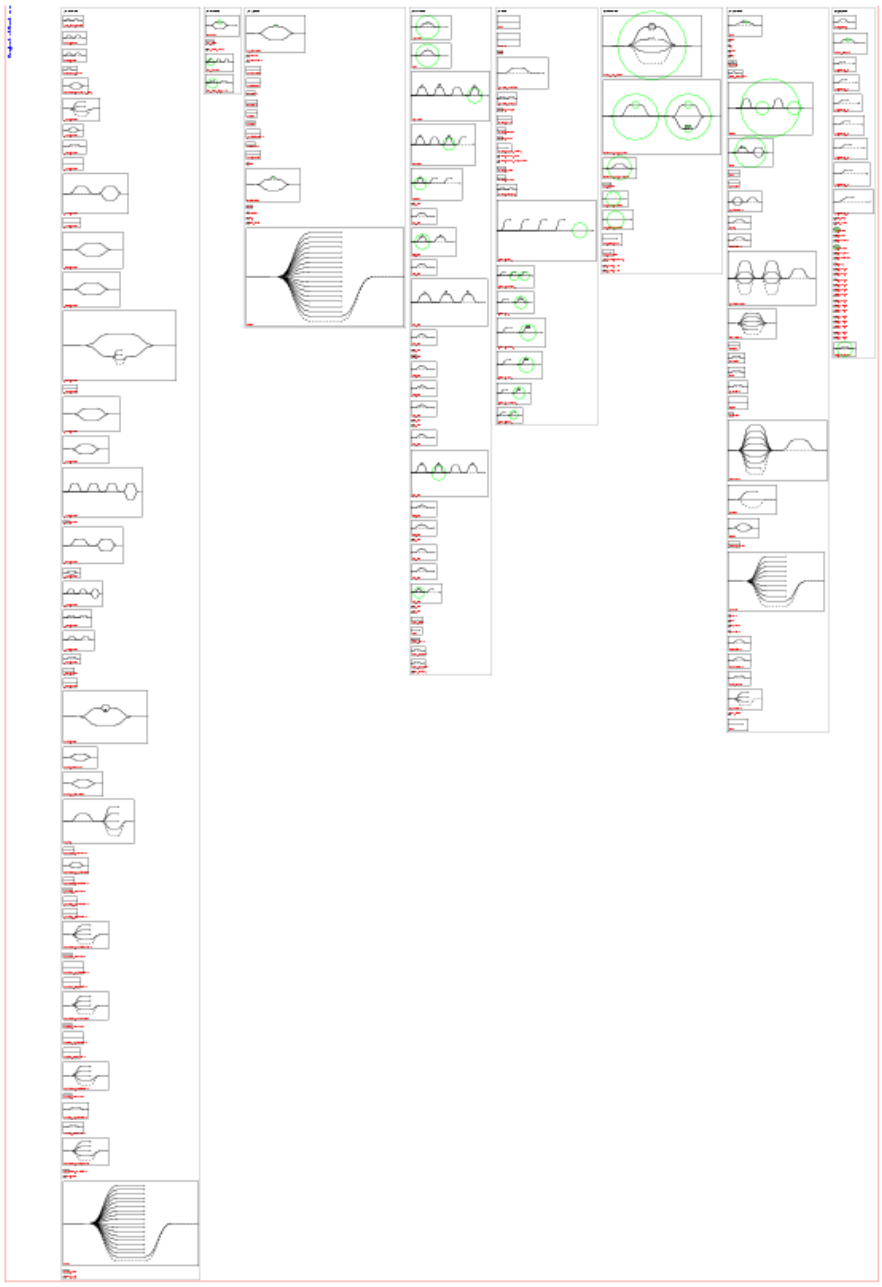


Figure 5.3: Map of Pico v3.

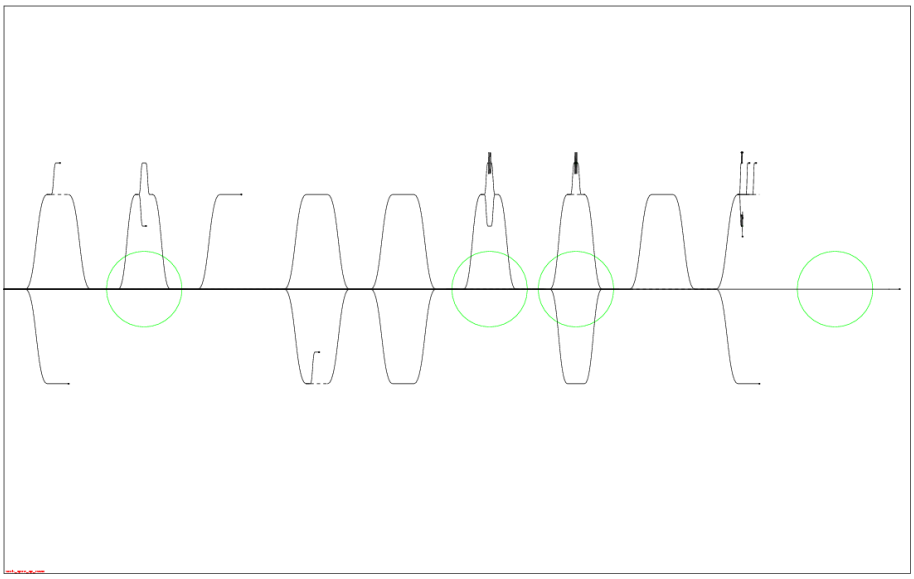


Figure 5.4: Example of mapped control flow.

5.2.2 Results

The results that we can get from the visualisation tool can be seen in figure 5.3. This shows a visualisation of the source code for the interpreter of Pico[8], a programming language developed at the Vrije Universiteit Brussel. Each vertical bar represents one source file. Each box within a source file represents a procedure definition. The size of each procedure box is determined by the number of statements in that procedure.

Figure 5.4 shows an example visualisation of such a procedure. As can be seen here, the control flow is charted within this box. Execution starts on the left hand side, and flows to the right. Branch points indicate branch points in the control flow, for instance due to conditional statements. Circles indicate repetition of the subpath, for instance due to for-loops. From this we can already visualise the complexity of procedures, as well as make an assessment of which procedures/modules are most important. We have also been able to use it to identify basic code duplication, which presents similar visualisations (see figure 5.5 as an example).

Future plans for this tool include the visualisation of join points and pointcuts, as well as the addition of more complex metrics and layouts.

[8] http://pico.vub.ac.be/

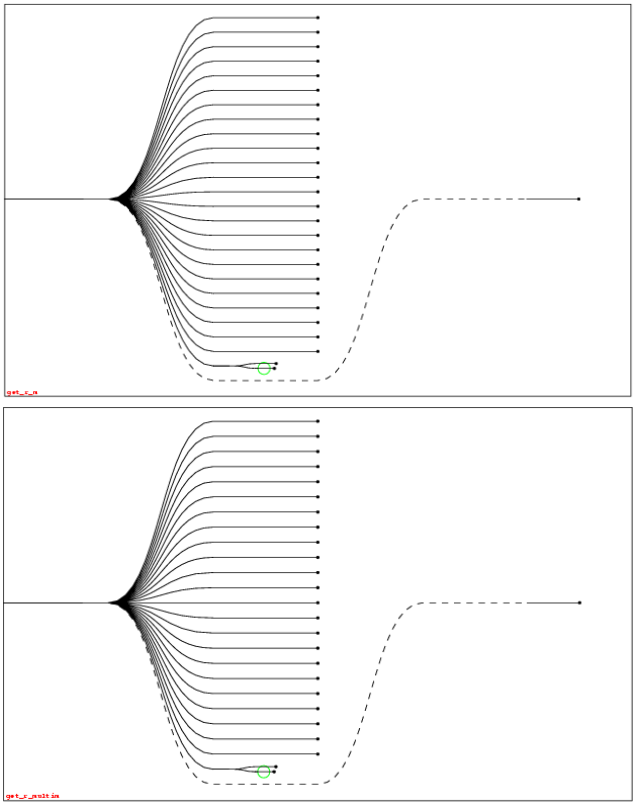


Figure 5.5: Possible code duplication.

The latter is intended to find visualisations which can be used to convey more information while maximising cognition.

5.2.3 Conclusion

The implementation of Runya has acted as another validation of the flexibility of the transformation framework as described in chapter 4. The choice for an XML representation of abstract syntax trees in this framework is what made the easy development and integration of Runya possible. In this it shows how the framework may be extended for purposes other than aspect weaving.

5.3 Aspect-enabled dynamic analysis

As a first example of what Aspect-oriented Programming and Logic Meta-programming makes possible, we will now describe how it helped to enable dynamic analyses in a real-life case study. This study was performed in cooperation with the LORE[9] research lab from the University of Antwerp.

The focus of this section will lie on the support we can get from AOP and LMP, not on the dynamic analyses themselves. More information on the case study in its entirety can be found in [ZAD+06, ZAD05, ADZ05].

5.3.1 The environment

The industrial partner that we cooperated with in the context of this research experiment is Kava[10]. Kava is a non-profit organization that groups over a thousand Flemish pharmacists. While originally safeguarding the interests of the pharmaceutical profession, it has evolved into a full fledged service-oriented provider. Among the services they offer is a tarification service —determining the price of medication based on the patient's medical insurance. As such they act as a financial and administrative go-between between the pharmacists and the national healthcare insurance institutions.

[9] *Lab On REengineering* (`http://www.lore.ua.ac.be/`)

[10] KAVA, **K**oninklijke **A**pothekers**V**ereniging van **A**ntwerpen, `http://www.kava.be/`. See also section 1.1.1.

Kava was among the first to realize the need for an automated tarification process, and have taken it on themselves to deliver this to their members. Some 10 years ago, they developed a suite of applications written in non-ANSI-C for this purpose. Due to successive healthcare regulation changes and forthcoming changes in the dataflow, they feel the necessity to reengineer their applications.

Kava has ported their application portfolio to a fully ANSI-C compliant version, running on Linux. Over the course of this migration effort it was noted that documentation of the application was outdated. This provided us with the perfect opportunity to undertake a re-documentation experiment, thus tackling the pitfall described in section 1.1.5.

5.3.2 The case study

Recently a number of novel dynamic analysis techniques that deal with program comprehension have been developed [GD05, HLBAL05, ZD04, ZCDP05]. Most of these techniques have been explored in the context of Object Orientation, but we considered it worthwhile to verify whether these techniques could be "transplanted" to the context of procedural systems.

As a scenario to test our dynamic analysis, the developers at Kava referred to the so-called "TDFS"[11] application. This is a batch-like application, which produces a digital and detailed invoice of all prescriptions for the healthcare insurance institutions. It is the end-stage of a monthly control- and tariffing process, and also acts as a control-procedure as the results are matched against the aggregate data that is collected earlier in the process. As such, it is used as a final check to see whether adaptations in the system have any unforeseen consequences —a form of regression testing.

5.3.3 Our approach

To analyse the dynamic behaviour of the TDFS application, a detailed trace of some representative runs of the system was required. To this end we applied a simple tracing aspect for instrumentation of the code.

[11] TDFS (Dutch): TariferingsDienst Factuur en Statistiek

Figure 5.6, shows part[12] of this aspect[13], written in Aspicere[14].

The idea is to trace calls to all procedures except for the `printf`- and `scanf`-families (line 4), and stream tracing information into a file (`fp`, declared on line 1) before and after each call (lines 10 and 17). Opening and closing of the file pointer on line 1 is achieved by advising the `main`-procedure (line 24).

In figure 5.6, LMP is used to parameterize the return type of the advised procedure calls: `Type` is bound on line 5, and subsequently used in the advice's signature (line 3) as well as in its bodies (line 7). This way, the `tracing` advice is not limited to one particular type of procedures. The information needed for the trace is accessed through a `thisJoinPoint` construct, similar to those in AspectJ-like languages, which is really a join point-specific binding (`Jp` on lines 3 and 23) and used as such (lines 11 and 18).

Applied to reverse-engineering contexts, using AOP, LMP and a template mechanism allows non-invasive and intuitive extraction of knowledge hidden inside legacy systems, *without* prior investigation or exploration of the source code. One does not first have to extract all available types and duplicate the tracing advice for each of them, as was experienced in [BvDT04].

5.3.4 Interference from the build system

When applying this to the Kava case study, we encountered one major obstacle: the build system. Aspicere's weaver transforms base code and aspects into woven C code, and as such acts as a pre-processor to a normal C compiler. Because the original makefile hierarchy drives the production of object files, libraries and executables, using a myriad of other tools and pre-processors (e.g. embedded SQL), and all of these potentially process advised input, it turns out that Aspicere's weaver crosscuts the makefile system.

To overcome this, we needed to find out what is produced at every stage of the build and unravel accompanying linker dependencies.

[12] We do not show advice for void procedures, as these are equivalent to the advices shown, less the need for a temporary variable to hold the return value.

[13] Note that the pointcut uses `call(Jp,"^(?!.*printf$|.*scanf$).*$")`, rather than `!call(Jp,"printf") and !call(Jp,"scanf")`. This is because Aspicere does not really support the second version, as it will only bind `Jp` through the `call` predicate. It is not bound by the `on (Jp)`, and so the negation of calls would fail.

[14] Aspicere was dicussed in section 5.1.

```
1 static FILE*  fp;

3 Type around tracing (Type) on (Jp):
    call(Jp,"^(?!.*printf$|.*scanf$).*$")
5   && type(Jp,Type) && !is_void(Type)
  {
7   Type i;

9   /* call sequence */
    fprintf (fp, "before␣(␣%s␣in␣%s␣)\n",
11       Jp->functionName, Jp->fileName);

13  /* continue normal control flow */
    i = proceed ();
15
    /* return sequence */
17  fprintf (fp, "after␣(␣%s␣in␣%s␣)\n",
        Jp->functionName, Jp->fileName);
19
    return i;
21 }

23 int around cleanup () on (Jp):
    execution(Jp,"main")
25 {
    int i;
27
    /* open logfile */
29  fp = fopen ("mylog.txt", "a");

31  /* continue with main program */
    i = proceed ();
33
    /* close logfile */
35  fclose (fp);

37  return i;
  }
```

Figure 5.6: Part of the tracing aspect for enabling dynamic analyses.

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

Figure 5.7: 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
```

Figure 5.8: Adapted makefile.

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

Figure 5.9: 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
```

Figure 5.10: Adapted *esql* makefile.

This requires an inventory of the included tools, and their interplay. For each of these we must then find a way to plug in the weaver. More specifically, Aspicere's weaver needs one (pre-processed) input at a time and its output will be another tool's input. Additionally, the normal weaving habit is to transform aspects into genuine C compilation units by converting the advices into (multiple) procedures. This enables the normal C visibility rules in a natural way, i.e. the visibility of `fp` on figure 5.6 is tied to the module containing the aspect. To accomplish this modularisation, we need to link this single transformed aspect into each advised application.

In case all makefiles are automatically generated using, for instance, automake, one could try to replace (i.e. alias) the tools in use by wrapper scripts which invoke the weaving process prior to calling the original tool. The problem here is that this is an all-or-nothing approach. It may be that in some cases weaving is needed (e.g. a direct call to `gcc`, as in figures 5.7 and 5.8), and in others not (e.g. when `gcc` is called from within `esql`, as in figures 5.9 and 5.10). Making the replacement smart enough to know when to do what is not a trivial task.

In our case, many of the 267 makefiles were indeed generated. Still, some were manually adapted afterwards, while others were written

Module	Value
e_tdfs_mut1.c	0.81
tdfs_mut1_form.c	0.45
tdfs_bord.c	0.40
tdfs_mut2.c	0.16
tools.c	0.16
io.c	0.13
csrout.c	0.03
tarpargeg.c	0
csroutines.c	0
UW_strncpy.c	0
td.ec	0
cache.c	0
decfties.c	0
weglf.c	0
get_request.c	0

Figure 5.11: Results of the webmining technique.

from scratch. Due to issues with the embedded SQL pre-processor and the irregular presence of certain environment variables, we wrote some scripts to directly alter the makefiles and "weave in" the right operations on Aspicere's weaver instead of aliasing the tools themselves. However, detecting where exactly our tool failed (due to the heterogeneously structured makefiles) and making the necessary manual adaptations still took several hours.

Without intimate knowledge of the build system, it was hard to tell whether source files are first compiled before linking all applications, or (more likely) whether all applications are compiled and linked one after the other. As such, our weaving approach was not viable. As an ad hoc solution, we opted to move the transformed advice into the advised base modules themselves. This meant that we had to declare the file pointer `fp` as a local variable of the `tracing` advice (see figure 5.12, line 8), resulting in huge run-time overhead due to repeated opening and closing of the file.

```
Type around tracing (Type) on (Jp):
  call(Jp,"^(?!.*printf$|.*scanf$).*$")
  && type(Jp,Type) && !is_void(Type)
{
  Type i;

  /* open logfile */
  FILE* fp = fopen ("mylog.txt", "a");

  /* call sequence */
  fprintf (fp, "before ( %s in %s )\n",
       Jp->functionName, Jp->fileName);
  fflush (fp);

  /* continue normal control flow */
  i = proceed ();

  /* return sequence */
  fprintf (fp, "after ( %s in %s )\n",
      Jp->functionName, Jp->fileName);

  /* close logfile */
  fclose (fp);

  return i;
}
```

Figure 5.12: Part of the tracing aspect applied to the Kava case study.

5.3.5 Results

Having applied the aspect in figure 5.12 to the 407 C modules of the Kava application, and having run the TDFS scenario as described earlier, we wound up with a trace file of over 90GB of data. This data was then processed by the analysis tools, which were successful in identifying the most important modules of the system. Figure 5.11 shows these results as they were extracted by a webmining technique. The top ranking is in agreement with that of the domain experts. More on these results can be found in [ZAD+06, ZAD05, ADZ05]. They are out of the scope of this dissertation.

5.3.6 Follow-up

So far we have been able to identify what are the most important modules of an application through our aspect-enabled dynamic analysis technique. While a step in the right direction, this information is still quite coarse-grained (i.e. collections of procedures, rather than individual procedures). We would like to look at ways to delve deeper into the code. We would like to have more detailed introspection of the code but only with respect to the previously identified modules.

We can achieve this goal by setting up our pointcuts to match only join points in these modules. We ignore modules which we consider unimportant; we are "slicing" away what is of no immediate interest. We therefore call this approach *aspect-enabled slicing*.

By extending our advice to be a lot more verbose, we can output information concerning parameters and return values. This, together with the control flow reasoning, should allow us to get a more detailed view of how the most important modules interact with their immediately collaborating modules. Once more detailed information has been learned we can again feed this into our slicing approach. By factoring in the newly gained information we can again extend the logic in our pointcuts to either delve even deeper, or to validate our information.

5.3.7 Conclusion

AOP allowed us to quickly and easily instrument a legacy application, and for it to output the required information on its runtime behaviour. It was thanks to LMP that this instrumentation aspect could be written

down concisely, and that we could access information about the execution context in a very straightforward manner. We have also shown that this can be used to enable several different dynamic analyses of existing legacy applications without the need for integrating various different tools.

5.4 Business rule mining

In [MDDH04], Isabel Michiels and the author discuss the possibility of using dynamic aspects for mining business rules from legacy applications. Some suggestions as to how this may be done are presented based on the following fictitious, though realistic case:

> "*Our accounting department reports that several of our employees were accredited an unexpected and unexplained bonus of 500 euro. Accounting rightfully requests to know the reason for this unforeseen expense.*"

We will now revisit this case, showing the actual Cobble advices which may be used to achieve the ideas set forth in [MDDH04]. In doing so we present another argument in favour of how AOP+LMP may help with the recovery of the design of an existing application.

5.4.1 Initial facts

We start off with the information present in the problem statement. The accounting department can give us a list of the employees which got "lucky" (or rather unlucky, as their unexpected bonus did not go by unnoticed). We can encode this knowledge as facts:

```
META-DATA DIVISION.
  FACTS SECTION.
    LUCKY-EID VALUE 7777.
    LUCKY-EID VALUE 3141.
    *> etc.
```

Furthermore, we can also find the definition of the employee file which was being processed, in the copy books:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
```

```
3 FILE CONTROL.
    SELECT EMPLOYEE-FILE ASSIGN TO EFILE.
5
  DATA DIVISION.
7 FILE SECTION.
    FD EMPLOYEE-FILE.
9   01 EMPLOYEE.
       05 EID PIC 9(4).
11     *> etc.
```

Lastly, from the output we can figure out, by looking at the output generating statements (i.e. `DISPLAY`), the name of the data item holding the total value. This data item, `BNS-EUR`, turns out to be an edited picture. From this we conclude that it is only used for pretty printing the output, and not for performing actual calculations. At some time during execution the correct value for the bonus was moved to `BNS-EUR`, and subsequently printed. E.g.:

```
MOVE MYSTERY-FIELD TO BNS-EUR.
*> ...
DISPLAY BNS-EUR.
```

So our first task is to find what variable that was.

5.4.2 Finding the right data item

We go at this by tracing all moves to `BNS-EUR`, but *only while processing one of our lucky employees*:

```
1 FIND-SOURCE-ITEM SECTION.
    USE BEFORE ANY STATEMENT
3   AND NAME OF RECEIVER EQUAL TO "BNS-EUR"
    AND BIND LOC TO LOCATION
5   AND IF EID EQUAL TO LUCKY-EID.
  MY-ADVICE.
7   DISPLAY EID, ": ", LOC.
```

In short, this advice states that before all statements (line 2) which have `BNS-EUR` as a receiving data item (line 3), and if `EID` (id for the employee being currently processed; see data definition higher up) equals a lucky id (runtime condition on line 5), we display the location of that statement as well as the current id.

The runtime condition on line 5 is similar to AspectJ's `if` condition. We could equally well have coded this aspect without it:

```
1 FIND-SOURCE-ITEM-ALT SECTION.
    USE BEFORE ANY STATEMENT
3   AND NAME OF RECEIVER EQUAL TO "BNS-EUR"
    AND BIND LOC TO LOCATION
5   AND BIND LUCKY TO LUCKY-EID.
  MY-ADVICE.
7   IF EID EQUAL TO LUCKY
      DISPLAY EID, ": ", LOC.
```

The semantics of this would have been identical to the previous example. The difference lies in the optimisations the weaver is able to perform. This last version will instantiate the advice for each lucky id, as the advice body is quantified in terms of this. The first version can prevent this as its advice body does not need this information, and we can therefore collect all run-time conditions and group them together.

Back to the experiment: we now find the possibilities to be one of several string literals (which we can therefore immediately disregard) and a variable named `BNS-EOY`, whose name suggests it holds the full value for the end-of-year bonus.

5.4.3 Checking the calculation

Our next step is to figure out how the end value was calculated. This allows us to check the figures and maybe detect an error. We set up another aspect to trace all statements modifying the variable `BNS-EOY`, but again only while processing a lucky employee. We do this in three steps. First:

```
  TRACE-BNS-EOY SECTION.
2   USE BEFORE ANY STATEMENT
    AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
4   AND BIND LOC TO LOCATION
    AND IF EID EQUAL TO LUCKY-EID.
6 MY-ADVICE.
    DISPLAY EID, ": statement at ", LOC.
```

Before execution of any statement (line 2) having `BNS-EOY` as a receiving data item (line 3), and when processing a lucky employee (line 5), this would output the location of that statement. Next:

```
1 TRACE-BNS-EOY-SENDERS SECTION.
    USE BEFORE ANY STATEMENT
3   AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
    AND BIND SENDING TO SENDER
5   AND BIND SENDING-NAME TO NAME OF SENDING
    AND IF EID EQUAL TO LUCKY-EID.
7  MY-ADVICE.
    DISPLAY SENDING-NAME, " sends ", SENDING.
```

This outputs the name and value for all sending data items (lines 4 and 5) before execution of any of the above statements. This allows us to see the contributing values. Lastly, we want to know the new value for `BNS-EOY` which has been calculated.

```
  TRACE-BNS-EOY-VALUES SECTION.
2   USE AFTER ANY STATEMENT
    AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
4   AND IF EID EQUAL TO LUCKY-EID.
   MY-ADVICE.
6   DISPLAY "BNS-EOY = ", BNS-EOY.
```

We now find a data item (cryptically) named `B31241`, which is consistently valued 500, and is added to `BNS-EOY` in every trace.

5.4.4 Verifying our assumption

Before moving on we would like to make sure we are on the right track. We want to verify that this addition of `B31241` is only triggered for our list of lucky employees. Again, a dynamic aspect allows us to trace execution of exactly this addition and helps us verify that our basic assumption holds indeed. We start by recording the location of the "culprit" statement as a usable fact:

```
  META-DATA DIVISION.
2   FACTS SECTION.
      CULPRIT-LOCATION VALUE 666.
4     *> other facts as before
```

The test for our assumption may then be encoded as:

```
  TRACE-BNS-EOY-SENDERS SECTION.
2   USE BEFORE ANY STATEMENT
    AND LOCATION EQUAL TO CULPRIT-LOCATION
```

```
4   AND IF EID NOT EQUAL TO LUCKY-EID.
  MY-ADVICE.
6   DISPLAY EID, ": back to the drawing board.".
```

This tests whether the culprit statement gets triggered during the process of any of the other employees. If it does, then something about our assumption is wrong. Or it may be that the accounting department has missed one of the lucky employees.

5.4.5 Rediscovering the logic

Given the verification that we are indeed on the right track, the question now becomes: why was this value added for the lucky employees and not for the others? Unfortunately, the logic behind this seems spread out over the entire application. So to try to figure this out we would like to have an execution trace of each lucky employee, including a report of all tests made and passed, up to and including the point where `B31241` is added. Dynamic aspects allow us to get these specific traces.

First, some preliminary work:

```
  WORKING-STORAGE SECTION.
2   01 FLAG PIC 9 VALUE 0.
       88 FLAG-SET     VALUE 1.
4      88 FLAG-NOT-SET VALUE 0.
```

The `FLAG` data item will be used to indicate when tracing should be active and when not. For ease of use we also define two "conditional" data items: `FLAG-SET` and `FLAG-NOT-SET`[15]. These reflect the current state of our flag.

Our first advice is used to trigger the start of the trace:

```
  TRACE-START SECTION.
2   USE AFTER READ STATEMENT
    AND NAME OF FILE EQUAL TO "EMPLOYEE-FILE"
4   AND BIND LOC TO LOCATION
    AND IF EID EQUAL TO LUCKY-EID.
6   MY-ADVICE.
```

[15] A negative flag is needed as one can not set a flag to `FALSE` in (legacy) Cobol. While this may feel like a silly restriction, the semantics of this are not as straightforward as may seem. The question is what the value of `FLAG` should become when `FLAG-SET` is set to false. It could be any value other than 1. But there may be more than one conditional data item reacting to this value. Which of those should then be made "true"? Modern day Cobol solves this problem by allowing a conditional data item to have a "`WHEN SET TO FALSE`" clause.

```
    SET FLAG-SET TO TRUE.
8   DISPLAY EID, ": start at ", LOC.
```

I.e., whenever a new employee record has been read (line 2 and 3), and that record is one for a lucky employee (line 5), we set the flag to true (line 7). We also do some initial logging (line 8).

The next advice is needed for stopping the trace when we have reached the culprit statement:

```
  TRACE-STOP SECTION.
2   USE AFTER ANY STATEMENT
    AND LOCATION EQUAL TO CULPRIT-LOCATION.
4  MY-ADVICE.
    SET FLAG-NOT-SET TO TRUE.
6   DISPLAY EID, ": stop at ", LOC.
```

Then it is up to the actual tracing. We capture the flow of procedures, as well as execution of all conditional statements:

```
  TRACE-PROCEDURES SECTION.
2   USE AROUND PROCEDURE
    AND BIND PROC TO NAME
4   AND BIND LOC TO LOCATION
    AND IF FLAG-SET.
6  MY-ADVICE.
    DISPLAY EID, ": before ", PROC, " at ", LOC.
8   PROCEED.
    DISPLAY EID, ": after ", PROC, " at ", LOC.
10
  TRACE-CONDITIONS SECTION.
12  USE AROUND ANY STATEMENT
    AND CONDITION
14  AND BIND LOC TO LOCATION
    AND IF FLAG-SET.
16 MY-ADVICE.
    DISPLAY EID, ": before condition at ", LOC.
18  PROCEED.
    DISPLAY EID, ": after condition at ", LOC.
```

From this trace we can then deduce the path that was followed from the start of processing a lucky employee, to the addition of the unexpected bonus. More importantly, we can see the conditions which were passed, from which we can (hopefully) deduce the exact cause.

5.4.6 Wrap-up of the investigation

This is where the investigation ends. We find that `B31241` is part of a business rule: it is a bonus an employee receives when he or she has sold at least 100 items of the product with number 31241. Apparently this product code had been assigned to a new product the year before. It was once associated to another product which had been discontinued for several years. The associated bonus was left behind in the code, and never triggered until employees started selling the new product. AOP and LMP provided us with a flexible and powerful tool to perform our investigation.

5.4.7 Conclusion

This section has shown how AOP+LMP can be used for rediscovering business logic in existing legacy applications, something which is of importance for the further evolution of such applications. To this end, we applied AOP and LMP in several ways: from smart and focused tracing, to verification of assumptions and, ultimately, the rediscovery of logic. It is exactly the flexibility of aspects, enhanced with LMP for reasoning about and reflecting on join points which makes it suitable for this task.

5.5 Encapsulating procedures

Business applications are faced, more and more, with the integration with other applications and services. One way to achieve this is by wrapping the original applications. In [SS03], Harry and Stephan Sneed propose creating web services from legacy host programs. They argue that while there exist tools for wrapping presentation access and database access for use in distributed environments,

> "*the accessing of existing programs, specifically the business logic of these programs, has not really been solved.*"

In an earlier paper, [Sne96a], Harry Sneed discusses a custom tool which allowed the encapsulation of Cobol procedures, to be able to treat them as "methods", a first step towards wrapping business logic. Part of that tool has the responsibility of creating a switch statement at the start of the program, which performs the requested procedure,

```
1 DISPATCHING SECTION.
    USE AROUND PROGRAM
3   AND BIND PARA TO PARAGRAPH
    AND BIND PARA-NAME TO NAME OF PARA
5   AND IF METHOD-NAME EQUAL TO PARA-NAME.
  MY-ADVICE.
7   PERFORM PARA.

9 ENCAPSULATION SECTION.
    USE AROUND PROGRAM.
11 MY-ADVICE.
    PERFORM ERROR-HANDLING.
13  EXIT PROGRAM.
```

Figure 5.13: Aspect for procedure encapsulation.

depending on the method name. This section shows how the same effect can be achieved using AOP and LMP, without the need for a specialized tool.

5.5.1 A basic wrapping aspect

Figure 5.13 shows how encapsulation of procedures (or "business logic") can be achieved, in a generic way, using AOP and LMP. The aspect shown here, written in Cobble, consists of two advices: one named `DISPATCHING`, the other `ENCAPSULATION`.

The first advice (lines 1–7) takes care of the dispatching. It acts around the execution of the entire program (line 2), and once for every paragraph in this program (line 3). The latter effect is caused by the ambiguousness of the `PARAGRAPH` selector. This can be any of a number of values. Rather than just picking one, what Cobble does is *pick them all*: the advice gets activated for every possible solution to its pointcut, one after the other (see also section 3.7.2). This is due to the backtracking semantics embedded in the pointcut language.

Furthermore, the `DISPATCHING` advice will only get triggered when `METHOD-NAME` matches the name of the selected paragraph (extraction of this name is seen on line 4). This is encoded in a runtime condition on line 5. Finally, the advice body, when activated, simply

calls the right paragraph (PERFORM statement on line 7).

The second advice (lines 9–13) serves as a generic catch-all. It captures execution of the entire program (line 10), but replaces this with a call to an error handling paragraph (line 12) and an exit of the program (line 13). The net effect is that whenever the value in METHOD-NAME does not match any paragraph name in the program, the error will be flagged and execution will end. This, together with the first advice, gives us the desired effect.

We are left with the question of where METHOD-NAME is defined, and how it enters our program. The answer to the first question is simply this: any arguments which get passed into a Cobol program from the outside must be defined in a *linkage section*. I.e.:

```
LINKAGE SECTION.
01 METHOD-NAME PIC X(30) VALUE SPACES.
```

Furthermore, the program division needs to declare that it expects this data item as an input from outside:

```
PROGRAM DIVISION USING METHOD-NAME.
```

5.5.2 Problems with introductions

This begs the question as to how the input parameter used in the previous subsection was inserted in an AOP-like way. Simply: it was *not*. We tacitly assumed our aspect to be defined *inside* the target program (a so-called "intra-aspect"), which dismissed the need for any added introduction mechanism. Of course, for a truly generic aspect (an "inter-aspect") we need to remedy this.

Definition of the METHOD-NAME data item is no big problem. We can simply define it within an aspect module, which, upon weaving, would extend the target program (modulo some alpha-renaming to prevent unintended name capture):

```
IDENTIFICATION DIVISION.
 ASPECT-ID. PROCEDURE-WRAPPING.

DATA DIVISION.
LINKAGE SECTION.
  01 METHOD-NAME PIC X(30) VALUE SPACES.
```

From this, it becomes pretty obvious that `METHOD-NAME` should be used as an input parameter. The concept of a linkage section makes no sense for an external aspect module, as an aspect will never be called in such a way. Indeed, we might even say that it *should not* be used that way. Therefore the appearance of a linkage section is a sufficient declaration of intent.

The hard part lies with the semantics of declaring extra input data items on another program. What do we expect to happen?

- Does the introduction of an input data item by the aspect replace existing input items in the advised program, or is it seen as an addition to them? Either way we are invalidating the original contract/interface expected by others.
- If it is added to them, then where does it go into the existing list of inputs? At the front? At the back?
- What happens when multiple aspects define such input items? In what order do they appear? Relying on a precedence rule seems a reasonable solution.
- How do we handle updating the sites where the woven program gets called? The addition of an extra input item will have broken these.

Consider the C/Java/...equivalent of this: what does it mean to introduce new parameters on procedures/methods? More to the point, *should* we allow this? Sofar, the above example is the only case in which it is required.

5.5.3 A full encapsulation aspect

The complexity of encapsulating Cobol procedures increases when we consider another important feature of Sneed's tool (ignored until now):

> "*For each [encapsulated] method a data structure is created which includes all variables processed as inputs and outputs. This area is then redefined upon a virtual linkage area. The input variables become the arguments and the output variables the results.*" [Sne96a]

```
  { IDENTIFICATION DIVISION.
46      ASPECT-ID. PROCEDURE-WRAPPING.

48    DATA DIVISION.
      LINKAGE SECTION.
50    01 METHOD-NAME PIC X(30) VALUE SPACES. },

52 findall(
      [Name, Para, Wss],
54    ( paragraph(Name, Para),
        slice(Para, Slice),
56      wss(Slice, Wss)
      ),
58    AllInOut
  ),
60
  max_size(AllInOut, VirtualStorageSize),
62 { 01 VSPACE PIC X(<VirtualStorageSize>). },

64 all(member([Name, Para, Wss], AllInOut), (
      {  01 SLICED-<Name> REDEFINES VSPACE.},
66    all( (record(R, Wss), name(R, RName)), (
        clone_and_shift(R, "<RName>-<Name>", SR),
68      { <SR> }
      ))
70 )),
```

Figure 5.14: Full procedure encapsulation (part 1).

```
72 { PROGRAM DIVISION USING METHOD-NAME, VSPACE.
     DECLARATIVES. },
74
  all(member([Name, Para, Wss], AllInOut), (
76   { WRAPPING-FOR-<Name> SECTION.
         USE AROUND PROGRAM
78       AND IF METHOD-NAME EQUAL TO "<Name>".
        WRAPPING-BODY.
80   },
     all( (top_record(R, Wss), name(R, RName)),
82     { MOVE <RName>-<Name> TO <RName>.}
     ),
84   {    PERFORM <Name>.}
     all( (top_record(R, Wss), name(R, RName)),
86     { MOVE <RName> TO <RName>-<Name>.}
     )
88 )),

90 { ENCAPSULATION SECTION.
       USE AROUND PROGRAM.
92    MY-ADVICE.
       PERFORM ERROR-HANDLING.
94     EXIT PROGRAM.
     END DECLARATIVES. }
```

Figure 5.15: Full procedure encapsulation (part 2).

Put another way, we must find all data items on which the encapsulated procedures depend. These are then gathered in a new record (one per procedure), which redefines a "virtual linkage area" (in C terms: a union over all newly generated typedefs). This linkage area must then also be introduced as an input data item of the whole program. Such a requirement seems far out of the scope of AOP. While it has a cross-cutting concern in it (cfr. "for *each* method"), this concern can not be readily defined using existing AOP constructs.

Instead, figures 5.14 and 5.15 show a different approach to the problem. It is encoded neither in Cobble or Aspicere, opting for a different view on the AOP+LMP equation. Whereas the previous examples were based on LMP embedded in AOP, the new example is based on using a generative programming technique, similar to the approach in [BMD02, DD99a, ZHS04].

The code can be read as follows. Whatever is enclosed in curly brackets ($\{\ldots\}$) is (aspect-)code which is to be generated. This can be further parameterized by placing variables in "fishgrates" ($<\ldots>$), which will get expanded during processing. Everything else is Prolog, used here to drive the code generation.

Let us apply this to the code in figures 5.14 and 5.15. Lines 1 and 2 declare the header of our aspect, while lines 4–6 define the linkage section as discussed before. Lines 8–15 calculate all slices (`slice/2` on line 11) for all paragraphs (`paragraph/2` on line 10). From each of these we extract the working-storage section (`wss/2` on line 12), which gives us the required in- and output parameters, collected in `AllInOut` (line 14). From this we extract the size of the largest one (`max_size/2` on line 17) which is used next in the definition of the virtual storage space (line 18).

Next, for each paragraph (i.e. for each member of `AllInOut`), we generate a redefinition of the virtual space to include all data items on which that paragraph depends (lines 20–26). The redefinition can be seen on line 21, where it is given a unique name (i.e. `SLICED-`*paragraph-name*). Its structure is defined by going over all records in the working-storage section for that paragraph (line 22), cloning each record under a new, unique name while updating the level number[16] (line 23), and then outputting this new record (line 24). This concludes the data definition.

[16] Level numbers are used to indicate nesting of records. As existing records, which may have been top level records, must now be nested inside another, their level numbers must be updated.

Next, the procedure division is put down, declaring the necessary parameters (line 28). We then generate advice similar to that in figure 5.13, but now they need to perform some extra work. First, they must transfer the data from the virtual storage space as redefined for the paragraph, to the original records defined for the program (lines 37–39). The original paragraph may then be called without worry (line 40). Afterwards, the calculated values are retrieved by moving them back to the virtual storage space, again as redefined for the paragraph (lines 41–43). All that is left is the generic catch-all (lines 46–50), and the closing of the aspect (line 51).

Despite the inherent complexity of the problem, AOP+LMP allowed us to write down our crosscutting concern. LMP was leveraged to define our aspect by reasoning over the program. AOP was leveraged to tackle the actual weaving semantics, unburdening us from writing program transformations. Granted, we made use of a slicing predicate to do most of the hard work (line 11). Still, the use of libraries which hide such algorithms is another bonus we can get from LMP.

5.5.4 Conclusion

Business applications are faced with ever greater demands for integration with other systems. One approach to solving this problem is by means of wrapping the original applications. In this section we have shown how AOP+LMP is flexible enough to allow this, without the need for specialized tools. We have, however, had to take a step back when we found that merely embedding LMP within the pointcut language proved not flexible enough to handle the full encapsulation approach. Indeed, while this technique is simple, and allows greater flexibility in defining behaviour, it is not able to handler the definition of crosscutting data structures. By embedding AOP within LMP and, in essence, by turning the AOP+LMP equation into a code generation scheme, we can handle these definitions.

The choice of whether to use LMP embedded in AOP, or AOP embedded in LMP, is a pragmatic one. While the second approach is the stronger one, it is also the more complex one. We therefore argue that it is best to choose the first, unless it can not handle the required crosscutting (as in the case of crosscutting data definitions).

5.6 Year 2000 syndrome

The Y2K-bug is probably the best-known example of problems related to legacy systems. It nicely shows how applications are sometimes forced to evolve. It is important to understand that at the heart of this was not a lack of technology or maturity thereof, but rather the understandable failure to recognize that code written as early as the sixties would still be around some forty years later.

So might AOP+LMP have helped us out? The problem statement certainly presents a crosscutting concern: whenever a date is accessed in some way, make sure the year is extended.

5.6.1 Finding dates

This presents our first problem: how do we recognize data items for dates in Cobol? While Cobol has structured records, and stringent rules for how data is transferred between them, they carry no semantic information whatsoever. Knowing which items are dates and which are not requires human expertise. The nice thing about LMP is that we could have used it to encode this. E.g., using the meta-data structure presented in section 3.7.3:

```
META-DATA DIVISION.
  FACTS SECTION.
    DATE-ITEM VALUE DATE-DUE.
    DATE-ITEM VALUE DEADLINE.
    *> etc.
```

In C, where a disaster is expected in 2038[17] (hence Y2K38) due to the overflowing of the 32-bit time-format, the recognition problem is less serious because of C's more advanced typing mechanisms. A date in (ANSI-)C could be built around the standard time provisions (in "time.h"), or otherwise some (hopefully sensibly named) custom typedef. In the former case, recompiling the source code on a system using more than 32 bits to represent integers solves everything immediately. Whereas all variables in Cobol have to be declared in terms of the same, low-level Cobol primitives, C allows variables to be declared as instances of user-defined types. In this sense, the latter case (custom date type) represents much less of a problem. The check for a date

[17] More details on `http://www.merlyn.demon.co.uk/critdate.htm`

would be equivalent to a check for a certain type.

5.6.2 Manipulating dates

There is a second problem for tackling the Y2K problem in Cobol: given the knowledge of which data items carry date information, how do we know which part encodes the year? It may be that some item holds only the current year, or that it holds everything up to the day. A data item may be in *Gregorian* form (i.e. "yyddd") rather than standard form ("yymmdd"). Of course, that "standard" may vary from locale to locale (the authors would write it as "ddmmyy"). But again, we could use LMP to encode this knowledge.

Let us assume we can check for data items which hold dates, and that these have a uniform structure (in casu "yymmdd"). Then we might write something like:

```
1 AN-YYMMDD-FIX SECTION RETURNING MY-DATE.
    USE AROUND SENDING-DATA-ITEM
3   AND SENDING-DATA-ITEM IS DATE.
  MY-ADVICE.
5   MOVE PROCEED TO MY-DATE(3:8).
    IF MY-DATE(3:4) GREATER THAN 50 THEN
7     MOVE 19 TO MY-DATE(1:2)
    ELSE
9     MOVE 20 TO MY-DATE(1:2).
```

This is a variation on Cobble's around advice. It differs in that it may return a value (`RETURNING`-clause on line 1). It also differs in the sense that normal Cobble advice acts on the statement level, whereas the above advice must apply to individual getters (or sending data items, picked out by the `SENDING-DATA-ITEM` condition on line 2). As it captures sending data items the `PROCEED` statement is used in that fashion as well (line 5). Apart from this the advice is written as usual. In this case the logic is based on a date expansion scheme using a century window. I.e. all dates with a year greater than 50 are seen as lying in the 20th century. The others are bumped to the 21st century. While this just shifts the Y2K problem to a Y2K50 problem, it allows legacy applications to work with their updated, Y2K-ready friends.

5.6.3 Non-local data items

The previous advice has some problems though. One is the definition of `MY-DATE` (referred to as a return value on line 1, and assumed to have a "yyyymmdd" format). In Cobol, all data definitions are global. Hence, `MY-DATE` is a unique data item which gets shared between all advices. While this is probably safe most of the time, it could lead to subtle bugs whenever we have nested execution of such advice.[18] The same is true for all advices in Cobble. It is just that the need for a specific return value makes it surface more easily. Of course, in this case, the fix is to require duplication of this data item for all advice instantiations.

5.6.4 Weaving date access

The greater problem with the `AN-YYMMDD-FIX` advice lies in the weaving. When committed to a source-to-source approach, as we are with Cobble, weaving anything below the statement level becomes hard. As Cobol lacks the idea of functions[19], we can not replace access to a data item with a call to a procedure (whether advice or the original kind) as we could do in C. One remedy for this would be to divide up the original statements into subparts so that each data-item access becomes separated. This seeems hard, and we are not sure this is possible. Another possible remedy for would be to switch to machine-code weaving, but we are reluctant to do so, as we would lose platform independence. Common virtual machine solutions (e.g. as with ACUCobol) are not widespread either. The conclusion is, therefore, that the Y2K problem can not be tackled in Cobol using AOP and LMP, if the weaving is limited to a source-to-source approach.

5.6.5 Conclusion

This section has shown that the underlying language of a legacy application does, in fact, limit the expressivity of our AOP+LMP technique. We have seen that, in Cobol, we could not weave around accesses of data items. It is important to note the reason why this is so: because of our source-to-source approach to weaving. This entails that the woven

[18] Though not in this case, as the structure of the advice body *only* refers to the data item *after* the `PROCEED` statement.

[19] Functions can be written in later versions of Cobol. Our focus on legacy systems, however, rules these out for use here.

version of an application must be well-formed in the underlying language. In the case of Cobol this means that we must somehow wrap data item accesses in a way allowed by Cobol. Unfortunately, there is no way to do this in (legacy) Cobol. As a consequence, we had to take this into account in the design of the aspect language, which treats getters and setters as the statements (possibly) doing the getting or setting. These limitations are something which have to be considered for every language which one wants to extend using AOP, unless one is willing to move away from source-to-source weaving.

5.7 Conclusion

We have shown how AOP+LMP may be embedded in ANSI-C. This acts as another validation that this approach is flexible enough to be applied to very different languages (as Cobol and C are).

We have also demonstrated the flexibility of the transformation framework by enabling visualisations of legacy source code. These renderings of existing applications can help in regaining knowledge and understanding about them.

We then discussed four problems with legacy software, and showed how three of these might be aided through a mix of AOP and LMP. Tracing in C and business rule mining in Cobol went smoothly, using LMP as a pointcut mechanism in AOP. Encapsulation of procedures in Cobol required a more generative approach, by embedding AOP in LMP.

As for the Y2K problem in Cobol, only very advanced, nearly weaver-level pointcuts in synergy with various cooperating introductions might manage this. As it is, the semantics of Cobol, especially its lack of typing, presents too much of a limitation. In C, the Y2K38 problem can still be managed reasonably, precisely because it does feature such typing.

Chapter 6

Conclusions

It's a job that's never started that takes the longest to finish.

J. R. R. TOLKIEN

THIS dissertation has been arguing in favour of using a mix of Aspect-oriented Programming and Logic Meta-programming for the revitalisation of legacy software. The previous chapters have laid down the groundwork for this discussion. This chapter turns the attention back to the larger picture, and shows how everything fits together. We do so by re-iterating over the list of contributions of this dissertation, as it was declared in the introduction.

6.1 AOP for legacy environments

> "*We show that the Aspect-oriented Paradigm can be successfully embedded in legacy (non-OO) environments.*"

When talking about legacy environments, we mostly find ourselves in the realm of Cobol. This is corroborated by our findings within the ARRIBA research project [MDTZ03], as well as by the numbers put forward by the Gartner group: 75% of business data is processed in COBOL, with 180–200 billion LOC in use worldwide, and 15% of new applications written in COBOL. It is clear that Cobol still is *the* major player.

With only a limited market share left for the others, C does stand

out as an important legacy language. Again, when turning to the ARRIBA user committee, we find C is in active use, more so than other languages. Its popularity is also attested by other sources, such as the Tiobe Programming Community Index[1] and the analysis by DedaSys Consulting[2].

6.1.1 AOP for Cobol and C

This dissertation has covered the integration of AOP for both of these languages. In chapter 3 we presented *Cobble*, an aspectual extension for Cobol. Chapter 2 presented *Wicca*, an aspectual extension for C. Between the procedural paradigm of ANSI-C, and the statement-oriented ("English") paradigm of Cobol, we have shown our approach to be able to cover quite a broad spectrum of legacy languages.

6.1.2 AOP in legacy environments

What can we learn from this about AOP? Well, if AOP is *quantification and obliviousness* —as is argued in [FF05]— we find that AOP for legacy languages differs mainly in the kinds of quantification which can be made. I.e., rather than method execution we talk about procedures, which may not even have parameters or return values (cfr. Cobol). For lack of objects, there is also no talk about "sender" and "receiver". Despite these apparent simplifications, we do not necessarily end up with simpler languages: Cobol requires an extensive library of pointcuts in order to be able to quantify over all events of interest. Given the very different and much more extensive semantics of Cobol, this should come as no surprise.

As for obliviousness, there seems to be no impact on AOP. Rather the reverse, obliviousness in AOP brings extra modularity mechanisms to the table (namely aspects). This is interesting, especially for Cobol, which has only limited support for this —basically entire programs or nothing.

[1] `http://www.tiobe.com/tpci.htm`

[2] `http://www.dedasys.com/articles/language_popularity.html`

6.1.3 Weaving

On the weaving side, as explored in chapter 4, we have shown that a transformation framework based on XML representations can work, and work well at that. Through the XML format, and with a little help from the JVM, we were able to bring together a mix of disparate technologies and integrate them into a cohesive and functional whole (as can be seen in figure 4.3 of section 4.2). The hardest part lay in finding decent parsers for the legacy languages, which proved especially tricky in the case of Cobol (see section 4.3.1).

Transforming the XML representations is only limited in that the end-result of that transformation must be a valid program. This proved to be an obstacle for Cobol, where this well-formedness keeps us from weaving individual getters and setters (as discussed in section 3.5, as well as in section 5.6.4). Indeed, data items can not be replaced by anything other than data items. In C we can at least replace any reference to a variable with a call to a procedure. Not so in Cobol. Machine code weaving can hold the answer to this limitation for Cobol, but it would mean sacrificing our platform-independence —as well as push us far outside the scope of this dissertation.

6.1.4 Related work

AspectC and *AspectC++* were introduced in section 2.2. Apart from these two, there now also are:

- *Arachne* [DFL+05] is a dynamic weaver based on code splicing. Advised procedure calls are replaced by a jump to advice code. It features an expressive pointcut language (inspired by Prolog) and an extensible weaver (both for join point types as well as PCD's). A unique feature in Arachne are sequence-like pointcut definitions in which advice is just a C procedure.

- *TOSKANA* [EF05] applies the same technique of code splicing. Targeted mainly at autonomic computing, aspects are kernel modules woven in at run-time. The aspect language is very limited, as is the join point model.

- *TinyC* [ZJ03] implements advices as programs calling the Dyninst API (an instrumentation library) at the right join points. Sofar,

only call join points are supported, through regular expression matching.

- *C4* [FGCW05] attempts to replace a traditional patch system with a simplified AOP-driven, semantic approach. The programmer writes down advice in situ in the base program, without any quantification. The C4 unweaver extracts the changes into a separate C4 file (a semantic patch) which can then be distributed, and applied by others.

6.2 A need for LMP

> "*We show that logic-based pointcuts, in conjunction with access to a meta-object protocol, are able to fully enable Aspect-oriented Programming in legacy environments lacking reflective capabilities.*"

In order to write generic aspects we need to be able to reflect on the base programs, to extract information about that base program. With languages such as Java, which offer a reflection API natively, the aspect language need not do anything special —though there is research that questions this [GB03, KR05, De 01, HC03, HU03].

6.2.1 Make up for a lack of reflection

Cobol and C have no reflective capabilities, which indeed limits us from writing generic advice. This was experienced by Bruntink, Van Deursen and Tourwé in their work for ASML, the world market leader in lithography systems (see [BvDT04], as well as section 5.1.2 in this dissertation). The associated aspect languages must therefore make up for this. As we have argued in sections 1.2.4, 1.2.5, 1.2.6, 3.7, 3.9, and chapter 5, LMP, together with a simple template mechanism, can do this for us.

6.2.2 Make knowledge explicit

As we argued in section 1.2.4, the ability of LMP to record business knowledge can be of great advantage to legacy software. Where languages lack reflective capabilities, they must also lack the ability to en-

code information about programs. Hence, programmers are forced into hacks and workarounds.

During our discussion on business rule mining in section 5.4, we have demonstrated that LMP can be used —very simply— to encode such information. In cooperation with AOP we were then able to exploit this in a direct way.

6.2.3 Embedding AOP in LMP

When tackling the encapsulation of business logic (section 5.5) we encountered the limits of expressiveness when embedding LMP in AOP. We were not able to describe the complex, though crosscutting, data definitions required to solve the task at hand. When embedding LMP in AOP, we can extensively quantify behaviour, but we cannot quantify structure.

We therefore turned the AOP/LMP equation around. By embedding AOP in LMP, we can write down the complex crosscutting structures, as is demonstrated by figures 5.14 and 5.15 in section 5.5.3. This technique is close to the idea of parametric introductions, as in [HU03] and [BMD02].

6.2.4 Limited by underlying language

Cobol's support for data definitions is at once empowering and limiting. It allows for very detailed descriptions of anything from numbers to strings —consider having to write a number in Java/C/... which has exactly 11 decimals. Yet it does not allow the reuse of these definitions. As we noted in section 5.6, there is no easy way to figure out, for instance, which data items are dates. Any item holding six digits may be a date or it may not. Nor is it said that any item which is not made up of six or eight digits must therefore not be a date.

This again poses a limitation for writing generic advice: how do we quantify over types (e.g. dates) when there are none? The only solution seems to be that we should write down, as business knowledge, which data items pertain to a certain type. This is tedious at best.

6.3 Validation of our approach

> "*We argue that Aspect-oriented Programming, together with logic-based pointcuts, is an ideal tool for tackling major issues in business applications: recovery, restructuring, integration, evolution, etc.*"

In chapter 5, we discussed four problems with legacy software, and showed how these might be aided through a mix of AOP and LMP.

First, we showed how, through a simple tracing aspect, we managed to enable dynamic analyses in a real case study (section 5.3). LMP was used to make the tracing advice as generic as possible by overcoming a lack of reflection in the base language.

Second, we worked through a scenario for recovering business knowledge from a legacy application (section 5.4). AOP and LMP were applied here in several ways: from smart and focused tracing, to verification of assumptions and, ultimately, the rediscovery of logic.

Thirdly, we tackled the problem of encapsulating legacy applications using a generic aspect (section 5.5). A basic form of encapsulation could be done easily (as in figure 5.13), but the full-on approach required us to take a step back. Still, by placing AOP within LMP, we were able to tackle even this problem, and relatively straightforward at that (see figures 5.14 and 5.15).

Only in the fourth problem, Y2K, our source-to-source weaving approach was held back by the limitations of the base language (section 5.6). As it is, the semantics of Cobol, especially its lack of typing, present too much of a roadblock. In C, the Y2K38 problem can still be managed reasonably, precisely because it does feature such typing.

6.4 Hypothesis

> "*The combination of Logic Meta-programming and Aspect-oriented Programming aids in the recovery of business architectures from source code, as well as in the restructuring and integration of business applications.*"

By having embedded AOP and LMP in legacy languages, and hence in existing business environments, we now have at our disposal a flexible toolchain. There is no requirement to move away from the existing

development techniques; the toolchain can be used in addition to them, as we have seen in chapter 5.

Furthermore, through LMP we are able to express business concepts and architectural descriptions of business applications. This makes it possible to work with applications at a higher level of abstraction, which we hope will also encourage better architectural descriptions to emerge.

6.5 Future work

Further work remains to be done on two fronts. First, we should bring the aspect languages for C and Cobol up to an acceptable maturity. For C, this is now being done by Bram Adams in his work on Aspicere. This has already reached the stage where it was successfully applied to a real case study. For Cobol, a lot of work remains to be done. At the moment, Cobble only supports weaving of statements. The weaving of procedures remains to be done. Also, the querying and matching of Cobol structures remains to be completed. Given the complexities of Cobol, and the extent of its structures, this is not an easy task.

Secondly, given tools of a certain maturity, we need to do further validation of the AOP/LMP approach *in the field*. Sofar, we have only had one chance at doing this —the other examples being only academic discussions. Though successful, one sample does not make a trend.

Apart from this, we should also try to find out to what extent we need the AOP-embedded-in-LMP technique (see section 5.5.3) in our toolchain. This, again, is something which should become clear from real-life experimentation.

One more fundamental problem lies in the amount of reuse which the transformation framework allows. While it enables the integration, and therefore the possible reuse, of existing components, few of these can be directly applied to XML trees for code of differing languages. Indeed, querying and transforming of XML trees is tied to the structure thereof, and this is directly tied to the grammars for the original languages. As these are unlikely to match this means that the XML trees will be structured differently, which in turn will likely break any components working with this structure. Figuring out how we might increase the potential reuse here, for instance by implementing "language-agnostic" weavers, would be helpful.

Appendix A

Cobol

COBOL - A plump secretary.
She talks far too much,
and most of what she says can be ignored.
She works hard and long hours,
but can't handle really complicated jobs.
She has a short and unpredictable temper,
so no one really likes working with her.
She can cook meals for a huge family,
but only knows bland recipes.

FROM "PROGRAMMING LANGUAGES ARE LIKE WOMEN"

COBOL is English, but there are (a lot of) strict rules which govern its structure. This chapter gives a short introduction to this structure. The focus is not on completeness, but rather on giving the reader a feel for the language.

A.1 Main structure

All Cobol programs are made up of four "divisions". These are:

1. `IDENTIFICATION DIVISION.`
2. `ENVIRONMENT DIVISION.`
3. `DATA DIVISION.`

4. PROCEDURE DIVISION.

The order of these divisions is strict. They must appear as in the above sequence. Furthermore, divisions 2 and 3 are optional.

A.2 Identification division

The main goal of the identification division is very simple: to name the program. E.g.:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. MY-FIRST-COBOL-APPLICATION.
```

The rules for forming a program name (or any other name) are:

- Start with a letter.
- Use only letters, digits and hyphens.

One must however take care not to use one of the reserved keywords. Figures A.1 and A.2 show the reserved keywords for Fujitsu-Siemens Cobol 2000. There are over five hundred of these. To make matters worse, some are only reserved within a certain context.

A.3 Environment division

The environment division's main intent is to identify any disk files that are used by the program. For example:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT STUDENT-FILE ASSIGN TO "STUDENTS.DAT"
          ORGANIZATION IS LINE SEQUENTIAL.
```

This defines STUDENT-FILE to be a handle for the "students.dat" file (line 4), in which the data is stored in a sequential way, with newlines between each record (line 5).

There are many possible variations for identifying disk files. Here is one more:

```
ACCEPT, ACCESS, ACTIVE-CLASS, ADD, ADDRESS, ADVANCING, AFTER, ALL, ALLOCATE, ALPHABET,
ALPHABETIC, ALPHABETIC-LOWER, ALPHABETIC-UPPER, ALPHANUMERIC, ALPHANUMERIC-EDITED, ALSO,
ALTER, ALTERNATE, AND, ANY, ANYCASE, ARE, AREA, AREAS, AS, ASCENDING, ASSIGN, AT, AUTHOR,
B-AND, B-NOT, B-OR, B-XOR, BASED, BEFORE, BEGINNING, BINARY, BINARY-CHAR, BINARY-DOUBLE,
BINARY-LONG, BINARY-SHORT, BIT, BLANK, BLOCK, BOOLEAN, BOTTOM, BY, CALL, CANCEL, CBL-CTR,
CF, CH, CHARACTER, CHARACTERS, CHECKING, CLASS, CLASS-ID, CLOCK-UNITS, CLOSE, CODE,
CODE-SET, COL, COLLATING, COLS, COLUMN, COLUMNS, COMMA, COMMIT, COMMON, COMMUNICATION,
COMP, COMP-1, COMP-2, COMP-3, COMP-5, COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2,
COMPUTATIONAL-3, COMPUTATIONAL-5, COMPUTE, CONDITION, CONFIGURATION, CONSTANT, CONTAINS,
CONTENT, CONTINUE, CONTROL, CONTROLS, CONVERTING, COPY, CORR, CORRESPONDING, COUNT,
CREATING, CRT, CURRENCY, CURSOR, DATA, DATA-POINTER, DATABASE-KEY, DATABASE-KEY-LONG, DATE,
DATE-COMPILED, DATE-WRITTEN, DAY, DAY-OF-WEEK, DE, DEBUG-CONTENTS, DEBUG-ITEM, DEBUG-LINE,
DEBUG-NAME, DEBUG-SUB-1, DEBUG-SUB-2, DEBUG-SUB-3, DEBUGGING, DECIMAL-POINT, DECLARATIVES,
DEFAULT, DELETE, DELIMITED, DELIMITER, DEPENDING, DESCENDING, DETAIL, DISABLE, DISC, DISPLAY,
DIVIDE, DIVISION, DOWN, DUPLICATES, DYNAMIC, EBCDIC, EC, ELSE, ENABLE, END, END-ACCEPT,
END-ADD, END-CALL, END-COMPUTE, END-DELETE, END-DISPLAY, END-DIVIDE, END-EVALUATE, END-IF,
END-INVOKE, END-MULTIPLY, END-OF-PAGE, END-PERFORM, END-READ, END-RECEIVE, END-RETURN,
END-REWRITE, END-SEARCH, END-START, END-STRING, END-SUBTRACT, END-UNSTRING, END-WRITE, ENDING,
ENTRY, ENVIRONMENT, EO, EOP, EQUAL, EO, EOP, EQUAL, ERASE, ERROR, ESCAPE, EVALUATE, EVERY,
EXCEPTION, EXCEPTION-OBJECT, EXIT, EXTEND, EXTENDED, EXTERNAL, FACTORY, FALSE, FD, FILE,
FILE-CONTROL, FILLER, FINAL, FIRST, FLOAT-EXTENDED, FLOAT-LONG, FLOAT-SHORT, FOOTING, FOR,
FORMAT, FREE, FROM, FUNCTION, FUNCTION-ID, GENERATE, GET, GIVING, GLOBAL, GO, GOBACK, GREATER,
GROUP, GROUP-USAGE, HEADING, HIGH-VALUE, HIGH-VALUES, I-O, I-O-CONTROL, ID, IDENTIFICATION,
IF, IGNORING, IN, INDEX, INDEXED, INDICATE, INHERITS, INITIAL, INITIALIZE, INITIATE, INPUT,
INPUT-OUTPUT, INSPECT, INSTALLATION, INTERFACE, INTERFACE-ID, INTO, INVALID, INVOKE, IS,
JUST, JUSTIFIED, KEY, LABEL, LAST, LEADING, LEFT, LENGTH, LESS, LIMIT, LIMITS, LINAGE,
LINE, LINE-COUNTER, LINES, LINKAGE, LOCAL-STORAGE, LOCALE, LOCK, LOW-VALUE, LOW-VALUES,
MEMORY, MERGE, MESSAGE, METHOD, METHOD-ID, MINUS, MODE, MODULES, MORE-LABELS, MOVE, MULTIPLE,
MULTIPLY, NATIONAL, NATIONAL-EDITED, NATIVE, NEGATIVE, NESTED, NEXT, NO, NOT, NULL, NUMBER,
NUMERIC, NUMERIC-EDITED, OBJECT, OBJECT-COMPUTER, OCCURS, OF, OFF, OMITTED, ON, OPEN,
OPTIONAL, OPTIONS, OR, ORDER, ORGANIZATION, OTHER, OUTPUT, OVERFLOW, OVERRIDE, PACKED-DECIMAL,
PADDING, PAGE, PAGE-COUNTER, PERFORM, PF, PH, PIC, PICTURE, PLUS, POINTER, POSITION, POSITIVE,
PRESENT, PRINT-SWITCH, PRINTING, PROCEDURE, PROCEED, PROGRAM, PROGRAM-ID, PROGRAM-POINTER,
PROPERTY, PROTOTYPE, PURGE, QUOTE, QUOTES, RAISE, RAISING, RANDOM, RD, READ, RECEIVE, RECORD,
RECORDING, RECORDS, REDEFINES, REEL, REFERENCE, RELATIVE, RELEASE, REMAINDER, REMOVAL,
RENAMES, REPEATED, REPLACE, REPLACING, REPORT, REPORTING, REPORTS, REPOSITORY, RERUN, RESERVE,
RESET, RESUME, RETRY, RETURN, RETURNING, REVERSED, REWIND, REWRITE, RF, RH, RIGHT, ROLLBACK,
ROUNDED, RUN, SAME, SCREEN, SD, SEARCH,...
```

Figure A.1: Keywords for Fujitsu-Siemens Cobol 2000 (part 1).

```
..., SECTION, SECURITY, SEGMENT-LIMIT, SELECT, SELF, SEND, SENTENCE, SEPARATE, SEQUENCE,
SEQUENTIAL, SET, SHARING, SIGN, SIZE, SORT, SORT-MERGE, SORT-TAPE, SORT-TAPES, SOURCE,
SOURCE-COMPUTER, SOURCES, SPACE, SPACES, SPECIAL-NAMES, STANDARD, STANDARD-1, STANDARD-2,
START, STATUS, STOP, STRING, SUBTRACT, SUM, SUPER, SUPPRESS, SUPPRESSING, SYMBOLIC, SYNC,
SYNCHRONIZED, SYSTEM-DEFAULT, TABLE, TALLY, TALLYING, TAPE, TAPES, TERMINAL, TERMINATE,
TEST, THAN, THEN, THROUGH, THRU, TIME, TIMES, TO, TOP, TRAILING, TRUE, TRY, TYPE, TYPEDEF,
UNIT, UNITS, UNIVERSAL, UNLOCK, UNSTRING, UNTIL, UP, UPON, USAGE, USE, USER-DEFAULT, USING,
VAL-STATUS, VALID, VALIDATE, VALIDATE-STATUS, VALUE, VALUES, VARYING, WHEN, WITH, WORDS,
WORKING-STORAGE, WRITE, ZERO, ZEROES, ZEROS, ALL, AND, AS, B-AND, B-NOT, B-OR, B-XOR,
BYTE-LENGTH, CALL-CONVENTION, CHECKING, COBOL, DE-EDITING, DEFINE, DEFINED, DIVIDE, ELSE,
END-IF, END-EVALUATE, EQUAL, EVALUATE, FIXED, FLAG-85, FLAG-NATIVE-ARITHMETIC, FORMAT, FREE,
FUNCTION-ARGUMENT, GREATER, IF, IMP, IS, LEAP-SECOND, LESS, LISTING, LOCATION, MOVE, NOT,
NUMVAL, OFF, ON, OR, OTHER, OVERRIDE, PAGE, PARAMETER, PROPAGATE, SET, SIZE, SOURCE, THAN,
THROUGH, THRU, TO, TRUE, TURN, WHEN, ZERO-LENGTH
```

Figure A.2: Keywords for Fujitsu-Siemens Cobol 2000 (part 2).

```
1 ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
3 FILE-CONTROL.
      SELECT INVMAST ASSIGN TO "INVMASTI.DAT"
5             ORGANIZATION IS INDEXED
              ACCESS IS RANDOM
7             RECORD KEY IS IM-INDX.
```

This declares `INVMAST` (line 4) as an indexed file (line 5), which may be accessed in a random (i.e. non-sequential) way (line 6). The right record is selected based on the value in `IM-INDX`.

A short note on indentation: while for modern Cobol compilers indentation is of no matter, there was a time when this was not the case. Originally, Cobol code was entered on so-called "coding forms" (an example can be seen in figure A.3). These were limited to a width of 80 characters, where certain columns had special meanings. Columns 1–6, for instance, were regarded as whitespace, and were mainly used for line numbers. Column 7 was used to indicate comments (place an asterisk), or continuation lines (place a hyphen; useful when the previous line didn't fit within the limited space). Columns 8–71 was where the actual Cobol code could be written. This was again divided in two parts: column A and B, where certain statements could only appear in column B. Columns 72–80 were, again, treated as whitespace.

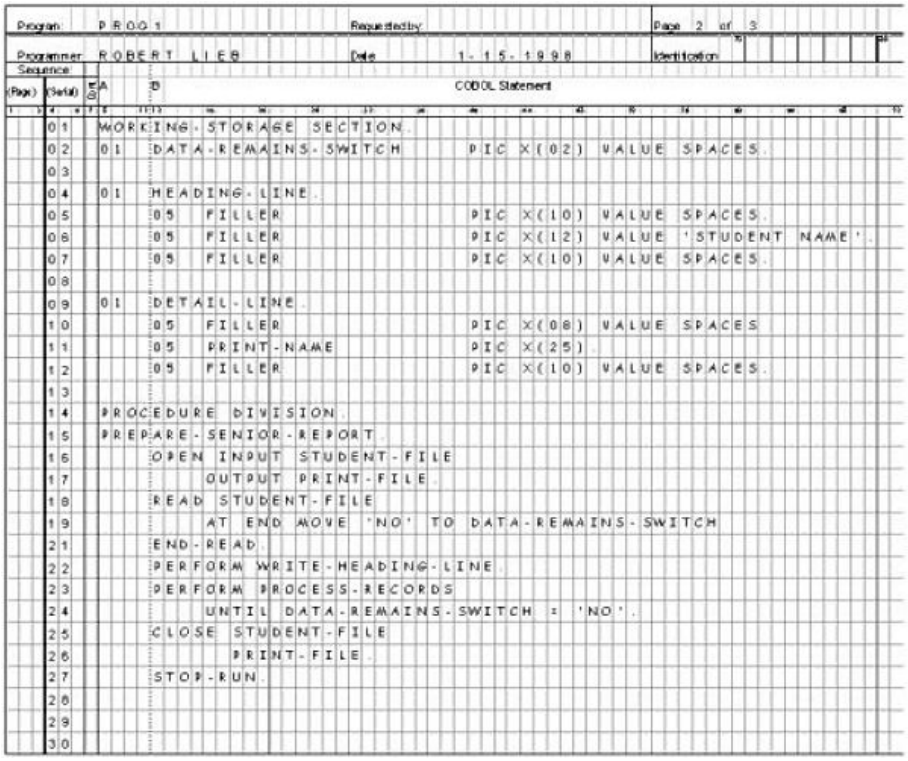


Figure A.3: Cobol coding form.

A.4 Data division

The data division is where one declares all data items (or variables) to be used in the program. There is one major rule here: *all data definitions are global*.

A data item itself is defined using the following ingredients:

1. Level number. This is used for defining nested records. If a data item is a top-level record, and has no subrecords, then this should be 77. Otherwise, choose a value in the range of 1–49.

2. Name. May be chosen freely (as long as it conforms to the naming rules), or it may be `FILLER`, which acts as a "don't care".

3. Picture. Defines the structure of the data. E.g. `999` would make it a number of 3 digits, `A(7)` a "string" of seven characters, etc.

4. An initial value. This may be the exact data, or one of the symbolic representations thereof: ZERO, ZEROES, SPACE, SPACES.

Some examples will clarify things:

```
77 STUDENT-NAME PIC A(64) VALUE SPACES.
```

This defines a data item named STUDENT-NAME which is made up of 64 characters, which will initially be spaces. Next:

```
01 STUDENT-RECORD.
  02 FIRST-NAME PIC A(32) VALUE SPACES.
  02 LAST-NAME  PIC A(64) VALUE SPACES.
```

This defines *one* record, which is named STUDENT-RECORD. It is made up of two subparts, FIRST-NAME and LAST-NAME. This is not because of the indentation, but due to the increased level numbers in their definition.

It is also possible for one data item to redefine another:

```
77 PERCENTAGE PIC ZZ9.9.
77 RATIO REDEFINES PERCENTAGE
          PIC X(4).
```

The first data item, PERCENTAGE, defines a decimal value of four digits. (The Z's indicate digits, which are not to be printed if they are zero.) RATIO redefines this data item as consisting of four alphanumeric characters. Both definitions, however, refer to the *exact same storage space*. That is, a change of one data item will be seen in the other. If you are familiar with the concept of unions in C, then this is pretty much the same.

Finally, data definitions may be placed in one of three sections:

- WORKING-STORAGE SECTION. This is for data which belongs to the program.
- FILE SECTION. This is where the record structure of files is defined.
- LINKAGE SECTION. This is for data items which may be passed to the program. Any Cobol program may be called from another Cobol program, and it is the data in the linkage section which allows them to communicate.

A.5 Procedure division

The procedure division is where we write down the program logic. It too has a very specific substructure. This is, from high-level to low-level:

- Sections.
- Paragraphs.
- Sentences.
- Statements.

It is the statements which define the low-level behaviour. A sentence starts with a verb, and may be followed by one or more clauses. The exact syntax differs for each verb. For instance, to print something on the display, one would write:

```
DISPLAY "HELLO␣WORLD"
```

To add two numbers, one might write:

```
ADD INTEREST TO AMOUNT
```

Alternatively we could have written:

```
COMPUTE AMOUNT = AMOUNT + INTEREST
```

Statements may be included in sentences. These are really nothing more than a bunch of statements followed by a dot. E.g.:

```
ADD 1 TO COUNTER
DISPLAY "Iteration␣#", COUNTER.
```

Some statements may also include other statements. The most obvious example of this is the conditional statement:

```
IF DO-NEW-ITERATION THEN
  ADD 1 TO COUNTER
  DISPLAY "Iteration␣#", COUNTER.
```

Note that in this case the dot is very relevant: it limits the scope of the `IF` statement. It is, however, better practice to write:

```
IF DO-NEW-ITERATION THEN
  ADD 1 TO COUNTER
  DISPLAY "Iteration␣#", COUNTER
END-IF
```

The `END-IF` keyword makes that we can dispense with the dot.

Sentences may be include in paragraphs or sections. Paragraphs may also be included in sections. A paragraph is nothing more than a labelled set of sentences. E.g.:

```
NEXT-ITERATION.
  ADD 1 TO COUNTER
  DISPLAY "Iteration␣#", COUNTER.
```

This defines a paragraph named `NEXT-ITERATION`. Similarly, a section is a labeled set of sentences and/or paragraphs. The label, however, is followed by a `SECTION` keyword, so as to differentiate it from paragraphs. E.g.:

```
NEXT-ITERATION SECTION.
  INCREMENT-COUNTER.
    ADD 1 TO COUNTER.
  DISPLAY-ITERATION.
    DISPLAY "Iteration␣#", COUNTER.
```

This shows a section named `NEXT-ITERATION`, consisting of two paragraphs, `INCREMENT-COUNTER` and `DISPLAY-ITERATION`, each of which has one sentence.

A.6 Control flow

Control flows from the top most statement, down to the bottom. Jumps of control flow can be done in several ways.

First, there is the `GO TO` statement. This jumps to the paragraph or section which is indicated by the argument, and starts executing from there. E.g.:

```
GO TO NEXT-ITERATION
```

The exact target may depend on a variable, as in:

```
GO TO NEXT-ITERATION
      END-OF-PROGRAM
   DEPENDING ON USER-INPUT
```

This would jump to either `NEXT-ITERATION` or `END-OF-PROGRAM` depending on the value of `USER-INPUT` (similar to a `switch` expression in Java).

As GO TO is considered harmful —rightfully so— there is another way of making jumps: PERFORM them.

```
PERFORM NEXT-ITERATION
```

This again starts executing from within the NEXT-ITERATION paragraph (or section). But, most importantly, when we reach the end of this procedure, the control flow returns to the statement immediately following the PERFORM statement.

There are several variations on the PERFORM statement for using it as a loop construct. E.g.:

```
PERFORM DO-ITERATION
  UNTIL END-OF-SESSION
```

This is a "do-while" loop. To turn it into a "while-do" loop we can write:

```
PERFORM DO-ITERATION
  UNTIL END-OF-SESSION
  WITH TEST BEFORE
```

We are not limited to jumping to paragraphs or sections. It is allowed to place the statements which should be looped directly into the PERFORM statement. E.g.:

```
PERFORM UNTIL END-OF-SESSION
        WITH TEST BEFORE
  ADD 1 TO COUNTER
  DISPLAY "Iteration␣#", COUNTER
  ACCEPT USER-INPUT
END-PERFORM
```

Note the different order of the clauses (here they are placed before the statements, rather than after the procedure name), as well as the compulsory END-PERFORM keyword.

Of course, there is also a variation which allows "for" loops:

```
PERFORM VARYING COUNTER FROM 1 BY 1
        UNTIL COUNTER EQUALS 10
  DISPLAY "Iteration␣#", COUNTER
END-PERFORM
```

Which would loop the DISPLAY statement ten times. When the number of iterations can be predetermined, as it can be here, we can also write:

```
PERFORM 10 TIMES
  ADD 1 TO COUNTER
  DISPLAY "Iteration␣#", COUNTER
END-PERFORM
```

Finally, a program ends when it has finished with the bottom-most statement, or when it encounters a `STOP RUN` or `EXIT PROGRAM` statement. There is a reason for the two different statements: the first will exit the entire run, whereas the second one will only exit from a subprogram, but will leave the calling program running. That means that the best way to end a Cobol program is:

```
EXIT PROGRAM
END RUN
```

This will exit the subprogram if this code was called as a subprogram. If not, then it will end the "run", i.e. do an exit to the operating system.

Bibliography

[ADZ05] Bram Adams, Kris De Schutter, and Andy Zaidman. AOP for legacy environments, a case study, 2005. European Interactive Workshop on Aspects in Software, EIWAS '05.

[Ame74] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language Cobol X3.23–1974*, 1974.

[Ame89] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, dec 1989.

[AT05] Bram Adams and Tom Tourwé. Aspect Orientation for C: Express yourself. In *SPLAT*, 2005.

[Bad00] G.J. Badros. JavaML: a markup language for Java source code. *Computer Networks*, 33(1-6):159–177, 2000.

[Ben95] Keith Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[BG96] Thomas J. Bergin and Richard G. Gibson. *History of Programming Languages*. ACM Press

[BMD02] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In *GPCE*, pages 110–127, 2002.

[Bow98] Antony Bowers. *Effective Meta-programming in Declarative Languages*. PhD thesis, Department of Computer Science, University of Bristol, January 1998.

[BS95] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems*. Morgan Kaufmann, 1995.

[BvDT04] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. An initial experiment in reverse engineering aspects. In *WCRE*, pages 306–307, 2004.

[CC90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, jan 1990.

[CK03] Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD*, pages 50–59, 2003.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *SIGSOFT Softw. Eng. Notes*, 26(5):88–98, 2001.

[DD99a] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd Int'l Conf. Reflection*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[DD99b] Maja D'Hondt and Theo D'Hondt. Is domain knowledge an aspect? In *ECOOP Workshops*, pages 293–294, 1999.

[DDMW00] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of the international symposium on Software Architectures and Component Technology 2000.*, 2000.

[DDN03] Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[De 01] Kris De Volder. Code reuse, an essential concern in the design of aspect languages? In *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, jun 2001.

[DFL+05] 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 *AOSD*, pages 27–38. ACM Press, 2005.

[DFW00] Kris De Volder, Johan Fabry, and Roel Wuyts. Logic meta components as a generic component model. In *Proceedings of the ECOOP '2000: Fifth International Workshop on Component-Oriented Programming*, 2000.

[DMW99] Maja D'Hondt, Wolfgang De Meuter, and Roel Wuyts. Using reflective programming to describe domain knowledge as an aspect. In *Proceedings of GCSE '99*, 1999.

[dOC98] Carlos Montes de Oca and Doris L. Carver. Identification of data cohesive subsystems using data mining techniques. In *ICSM*, pages 16–23, 1998.

[EF05] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05*, pages 51–62. ACM Press, 2005.

[FF05] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[FGCW05] Marc Fiuczynksi, Robert Grimm, Yvonne Coady, and David Walker. patch (1) Considered Harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD*, pages 60–69. ACM Press, 2003.

[GBNT01] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *CACM*, 44(10):87–93, 2001.

[GD05] Orla Greevy and Stephane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, pages 314–323. IEEE, 2005.

[GKMM04] Nicolas Gold, Claire Knight, Andrew Mohan, and Malcolm Munro. Understanding service-oriented software. *IEEE Softw.*, 21(2):71–77, 2004.

[Gol98] Nicholas Gold. The meaning of "legacy systems", 1998.

[GR04] J. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD 2004: Proc. of the 3rd International Conf. on Aspect-Oriented Software Development*, pages 36–45. ACM Press, 2004.

[HC03] Youssef Hassoun and Constantinos Constantinides. Visibility considerations and code reusability in AspectJ. In *3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society*, mar 2003.

[HH04] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD 2004: Proc. of the 3rd International Conf. on Aspect-Oriented Software Development*, pages 26–35. ACM Press, 2004.

[HL94] Patricia Hill and John Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[HLBAL05] Wahab Hamou-Lhadj, Edna Braun, Danien Amyot, and Timothy Lethbridge. Recovering behavioral design models from execution traces. In *CSMR*, pages 112–121, 2005.

[Hon98] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel, Departement of Computer Science, dec 1998.

[HU03] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *AOSD*, pages 80–89, 2003.

[ISO02] ISO/IEC. Information technology — Programming languages — COBOL, 2002. Reference number ISO/IEC 1989:2002(E).

[JC04] A. Jackson and S. Clarke. Sourceweave.net: Source-level cross-language aspect-oriented programming. In G. Karsai and E. Visser, editors, *Proc. of the 3rd International Conf. on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *LNCS*, pages 115–134. Springer, oct 2004.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.

[Kic97] Gregor Kiczales. Aspect-oriented programming. In *Proceedings of the Eighth Workshop on Institutionalizing Software Reuse*, 1997.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[KR05] Günter Kniesel and Tobias Rho. Generic aspect languages - needs, options and challenges. In *Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA)*. Sep 2005.

[Läm98] Ralf Lämmel. Object-oriented cobol: Concepts & implementation. In J. Wessler et al., editors, *COBOL Unleashed*. Macmillan Computer Publishing, sep 1998.

[LC03] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA 2003: Proc. of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12. ACM Press, 2003.

[Leh74] M. Lehman. Programs, cities, students, limits to growth? In *Imperial College of Science, Technology and Medicine Inaugural Lecture Series*, volume 9, pages 211–229, 1974.

[Leh96] M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, 1996.

[Leh98] M. M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, 1998.

[LS05a] Ralf Lämmel and Kris De Schutter. What does Aspect Oriented Programming mean to Cobol? In *AOSD '05*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[LS05b] Daniel Lohmann and Olaf Spinczyk. On typesafe aspect implementations in C++. In *SC '05*, 2005.

[LV01] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA*, pages 147–155, 1987.

[MDDH04] Isabel Michiels, Theo D'Hondt, Kris De Schutter, and Ghislain Hoffman. Using dynamic aspects to distill business rules from legacy code. In *DAW: Dynamic Aspects Workshop*, pages 98–102, mar 2004.

[MDTZ03] Isabel Michiels, Dirk Deridder, Herman Tromp, and Andy Zaidman. Identifying problems in legacy software: Preliminary findings of the ARRIBA project. In *ELISA workshop at ICSM 2003*, 2003.

[Men00] Kim Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 2000.

[MMW00] Tom Mens, Kim Mens, and Roel Wuyts. On the use of declarative meta programming for managing architectural software evolution. In *Proceedings of the ECOOP '2000 Workshop on Object-Oriented Architectural Evolution*, jun 2000.

[MMW02a] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *SEKE*, pages 289–296, 2002.

[MMW02b] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Syst. Appl*, 23(4):405–413, 2002.

[MN95] Gail C. Murphy and David Notkin. Lightweight source model extraction. In *SIGSOFT FSE*, pages 116–127, 1995.

[MT01] Tom Mens and Tom Tourwé. A declarative evolution framework for object-oriented design patterns. In *ICSM*, pages 570–579, 2001.

[MW03] Daniel L. Moise and Kenny Wong. An industrial experience in reverse engineering. In *WCRE*, pages 275–284, 2003.

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *ECOOP*, pages 214–240, 2005.

[PW92] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, oct 1992.

[Rit93] Denis M. Ritchie. The development of the C language. In *Proceedings of the Conference on History of Programming Languages*, volume 28(3) of *ACM Sigplan Notices*, pages 201–208, New York, NY, USA, apr 1993. ACM Press.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[Sam69] Jean E. Sammet. *Programming Languages: History and Fundamentals*. Prentice Hall, Englewood Cliffs, 1969.

[Sam78] Jean E. Sammet. The early history of COBOL. In *The first ACM SIGPLAN Conf. on History of programming languages*, pages 121–161. ACM Press, 1978.

[Sam85] Jean E. Sammet. Brief summary of the early history of COBOL. *j-ANN-HIST-COMPUT*, 7(4):288–303, oct /dec 1985.

[SGSp02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *TOOLS*, jan 2002.

[Sim00] S.E. Sim. Next generation data interchange: Tool-to-tool application program interfaces. In *Proc. of Working Conf. on Reverse Engineering (WCRE'00)*, pages 278–283. IEEE Computer Society, nov 2000.

[Sne96a] Harry M. Sneed. Encapsulating legacy software for use in client/server systems. In *WCRE*, page 104, 1996.

[Sne96b] Harry M. Sneed. Modelling the maintenance process at Zurich Life Insurance. In *ICSM*, page 217, 1996.

[Sne04] Harry M. Sneed. Program comprehension for the purpose of testing. In *IWPC*, pages 162–171, 2004.

[Sne05] Harry M. Sneed. An incremental approach to system replacement and integration. In *CSMR*, pages 196–206, 2005.

[SPS02] S. Schonger, E. Pulvermüller, and S. Sarstedt. Aspect-oriented programming and component weaving: Using XML representations of abstract syntax trees. In *Proc. of the 2nd German GI Workshop on Aspect-Oriented Software Development*, pages 59 – 64, feb 2002. Technical Report No. IAI-TR-2002-1, University of Bonn, Computer Science Dept.

[SS03] Harry M. Sneed and Stephan H. Sneed. Creating web services from legacy host programs. In *WSE*, pages 59–65, 2003.

[Ste05] Friedrich Steimann. Domain models are aspect free. In *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2005.

[Tas04] David Tas. Aspect-oriëntatie in Cobol, (graduation thesis, in Dutch), 2004.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, 2001.

[Vol98] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[Won04] Stijn Van Wonterghem. Aspect-orientatie bij procedurele programmeertalen, zoals C, (graduation thesis, in Dutch), 2004.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[ZAD05] Andy Zaidman, Bram Adams, and Kris De Schutter. Applying dynamic analysis in a legacy context: An industrial experience report. In *1st International Workshop on Program Comprehension Through Dynamic Analysis (PCODA)*, 2005.

[ZAD+06] 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 *CSMR*, pages 91–102, 2006.

[ZCDP05] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, pages 134–142. IEEE, 2005.

[ZD04] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pages 329–338, mar 2004.

[ZHS04] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating aspectJ programs with meta-aspectJ. In *GPCE*, pages 1–18, 2004.

[ZJ03] Charles Zhang and Hans-Arno Jacobsen. TinyC^2: Towards building a dynamic weaving aspect language for C. In *FOAL 2003*, Boston, MA, USA, 2003.

[ZK01] Y. Zou and K. Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *8th Asia-Pacific Software Engineering Conf. (APSEC 2001), 4-7 December 2001, Macau, China*, pages 390–499. IEEE Computer Society, 2001.