XOOTIC magazine

February 2007-Volume 12-Number 1

SOFTWARE TECHNOLOGY PDENG PROGRAMME

XOOTIC Symposium 2006 Proceedings

Contents

Advertorials

Colofon

XOOTIC MAGAZINE Volume 12, Number 1, February 2007

Editors: S. Estok, M.M. Lindwer, G. Gopakumar, L. Posta

XOOTIC Postal Address:
XOOTIC and XOOTIC MAGAZINE
P.O. Box 6122, 5600 MB Eindhoven, The Netherlands

E-mail board@xootic.nl *Internet* www.xootic.nl

OOTI Secretariat Postal Address:
Technische Universiteit Eindhoven
Department of Mathematics and Computer Science
c.o. Ms. M. de Wert
HG 6.57, P.O. Box 513, NL-5600 MB,
Eindhoven, The Netherlands

E-mail OOTI@tue.nl *Internet* http://wwwooti.win.tue.nl/

Printer Offsetdrukkerij De Witte, Veldhoven

XOOTIC SYMPOSIUM 2006

What could be the subject of a symposium that would not only attract XOOTIC members, but also students, researchers and people from industry? Since we are situated in Eindhoven, a place in the high-tech region of Europe involving triangle of Belgium- the Netherlands- Germany-, the straight answer to this question was evident: it should be the area of embedded systems. The presence of high-tech companies and research-institutes, and the fact that Eindhoven was going to become the European Design Capital 2006, have influenced the decision to move into the direction of embedded systems design.

However, it was challenging to be more specific, as there are many other symposia and conferences organized within this area. In order to differentiate the XOOTIC symposium from other symposia and conferences, we had intensive discussions to identify the most appealing topics. These discussions revealed that topics involving processes and methodologies that can be applied in the area of embedded systems could be engaging. From these different possibilities we filtered out topics related to methodologies (agile and model driven development), system development (component based development), software testing and validation and new paradigms (aspect oriented development). Hence, the title of the symposium came to the daylight **"Novel Approaches in Design and Architecture of Embedded Systems"** and the goal of the symposium was formulated: *to introduce recent trends in the design and architecture of embedded systems and to present current industrial alternatives to the XOOTIC members as well as the broader audience of industry and academia*.

Having setup our *mission* to offer attendees from different profiles a valuable symposium and having defined our *vision* to highlight novel approaches and their impact within embedded systems design, we challenged ourselves with determining the *strategy*.

It was a challenge because it is not only essential to determine the subject of the symposium and to find speakers that are able to realize that subject, but also to setup the complete organization of the symposium. It turned out to become a real-world project with real-world characteristics: dead-lines, scarce resources, changing conditions, organizational and communication challenges, etc. One of the first necessary pre-conditions for a success in such circumstances is a good team building. Throughout the whole trajectory of the organizational activities, we remained a team with a common goal. The support of the board as well as several other people was very valuable.

The speakers were prepared to travel from far places- Scott Ambler from Canada, Sagar Chaki from USA, and Uwe Assmann from Germany. All of them as well as Pierre van de Laar, our keynote speaker Egbert-Jan Sol, who corrected the topic of our symposium by accentuating the word "Architecting", and our chairman, Wim Hendriksen, were prepared to reserve some of their valuable time for this activity. The support of the OOTI management team, Harold Weffers and Maggy de Wert, during the whole period, especially during the last days of the organizational activities, was of enormous help. All of these have resulted in a beautiful celebration of the 15th anniversary of XOOTIC.

We would like to thank to all of you for your time and support for making this event thriving. We also want to thank all others who helped us, our sponsors and all of you who have joined us at the symposium day. We hope that it was a very valuable day for you!

It is a pleasure to offer you this magazine. We hope that it will serve you as a valuable material for making your knowledge gained during the symposium more concrete. Last but not least, we would like to thank the editor and magazine committee in helping us to bring out this Symposium Magazine.

XOOTIC Symposium Committee 2006,
Goce Naumoski, Yanja Dajsuren, Razvan Dinu, Ferdian Maswar, and Prashant Pandit.

Today your horizon is more than 10.000 km...

In het Nederlandse Veldhoven ontwikkelt ASML de snelste en nauwkeurigste ic-productiesystemen ter wereld. De klanten van deze uiterst geavanceerde machines – de grote chipfabrikanten – zitten verspreid over Europa, Azië en Noord-Amerika. ASML verzorgt de installatie, de training van gebruikers én de support. Vandaar dat altijd een deel van onze medewerkers klaarstaat om te vertrekken, onderweg is of zich – 10.000 kilometer verwijderd van Nederland – op locatie van de klant bevindt. Het succes van ASML is immers onlosmakelijk verbonden met het succes van onze klanten. Waar ook ter wereld.

...tomorrow it will be less than 45 nm.

Bij het ontwikkelen van onze ic-productiesystemen werken verschillende disciplines intensief samen in een uiterst dynamisch proces. Door die unieke samenwerking – door alle disciplines en geledingen heen – lukt het ons steeds weer om nóg snellere en nauwkeurigere systemen te produceren. Een structuur projecteren van 45 nanometer breed met een maximale positiefout van 8 nanometer bijvoorbeeld, is een ongekend grootse prestatie.

Om deze prestaties ook in de toekomst mogelijk te maken, zoeken wij gedreven technici die ons willen helpen die grenzen verder bij te stellen. Heb jij een afgeronde hbo-, wo-opleiding of PhD in een technische richting, bijvoorbeeld Werktuigbouwkunde, Mechatronica, Elektrotechniek, Softwaretechnologie, Natuurkunde of Technische Bedrijfskunde? En wil je jouw en onze grenzen verleggen? Kijk dan op **www.asml.com/toptechnici.** ASML zoekt zowel pas afgestudeerden als technici met ruime werkervaring.

ASML zoekt 200 toptechnici.

Kijk op www.asml.com/toptechnici

ASML
Commitment

Model Driven Architecture (MDA) and Component-Based Software Development (CBSD)

Prof. Dr. Uwe Aßmann, Technische Universität Dresden,
uwe.assmann@tu-dresden.de
http://st.inf.tu-dresden.de
http://www.rewerse.net/i3

In embedded software development, designers of product lines have to take both variations and extensions into account. Variations occur when modules are implemented differently or on different underlying architectures. Extensions are unplanned functional additions, resulting from product line evolution. This paper explores some universal concepts to combine both requirements.

Two major approaches to achieve variability and extensibility in a product line are model-driven architecture (MDA, by OMG) [MDA] and component-based software engineering (CBSE). Within MDA, the re-usable skeletons of applications are referred to as *Platform-Independent Models* (PIMs). A PIM captures the architecture and the algorithmic issues that are independent of all platforms. It is *translated* towards application models, specific for each execution platform and enriched by platform-specific information (Platform-Specific Models, PSM). These PSMs are then completed by hand towards the code of the products. The variability comes with the PSMs: the more PSMs are produced, the more products can be sold. Component-based software engineering (CBSE) serves the same goal. Here, *frameworks* and *components* play the role of PIM and PSM: a framework is instantiated towards an application by filling its *hooks* with components. However, although serving similar goals, both approaches differ in the way in which the application skeletons are instantiated: PIMs are translated towards applications; frameworks are linked, composed, or connected with components. Is there a way to combine both approaches? In other words, how to embed components into MDA, i.e., how to build, design and use MDA components?

Luckily, the way is not far, because MDA has a background in commonality/variability analysis. Taking a closer look, MDA is a design approach in which variability plays a major role: to build a product line, a PIM is extended to several PSMs, variants specific to a platform. Historically, the first approach to commonality/variability design has been Parnas' *information-hiding-based design* [Parnas]. In this approach, variabilities (design decisions that change) are separated into modules with fixed interfaces. When design decisions change, the implementations of these modules may change, without this having an impact on the interface. Clearly, this approach facilitates evolution, is robust against changes and well suited for product lines, since variants can be segregated into product-specific modules. However, Parnas' modular design method is based on *explicit composition interfaces,* which do not play any role in MDA.

This difference, however, can be explained, if *planned variability* in product lines is conceptually distinguished from *unforeseen extensibility* in software evolution. Clearly, a designer of a product line has knowledge where products vary, so that she can decide where variation points are inserted into a core framework, and which contracts guide their instantiation (*commonality/variability thinking*). On the other hand, software evolution is triggered by a customer who changes his requirements, and since such a change cannot be foreseen, the designer will not be able to plan how the software has to be *extended.* Hence, to prepare evolution, a designer also

needs to reflect about *stability/extension issues*. At least, this requires that a designer has to prepare for *implicit extension points* at which the skeletons *can potentially* be extended (also called *join points* [AOSD] or *implicit hooks* [ISC]). And this explains one difference between information-hiding based design and MDA: in Parnas' method, frameworks are varied at explicit variation points (interfaces), whereas in MDA, implicit extension points are employed.

These arguments lead to some interesting consequences. First of all, MDA is not only about platform issues, but rather about systematic variability. It is possible to base a PIM on templates, modules, and generic components, in short, all component models that use explicit variation points. With these techniques, a PIM can be varied towards products with systematic variations filling the explicit variation points – the degree of re-use depends only on the abstraction of the employed component model. Secondly, MDA can also be used for software evolution, if *grey-box* component models are employed that support unforeseen extension through implicit extension points. These new models, such as aspects [AOSD], hyperslices [HyperJ], role models [Roles], or fragment components [ISC] have been introduced to allow for merging and extension of components. With such a grey-box component model, a PIM can be extended by new components that are integrated at implicit extension points (join points). We also say that we *weave* an extension into a core model. With this grey-box technology, a PIM can be evolved in unforeseen ways, and MDA can be employed as an extension technology. Thus, in the future, there will be at least two major categories of MDA: the *parametric or generic MDA* for variability, based on black-box component models with explicit variation points, as well as the *extensible MDA* for evolution, based on grey-box component models with implicit extension points.

One problem remains: Who will build all the necessary tools, i.e. the template expanders and extension weavers for the multitude of specification and programming languages? Can we build template processors and weavers that work universally for all languages? Or, in other words, how can we build *universally generic* and *universally extensible languages*? Languages, that are suitable for universal templates and aspects? In the last years, our group has found a way to build grey-box component models for every language [REWERSE]. Given a metamodel of a language L, a fragment component model can be systematically generated for L, so that a re-use-oriented *add-on language Reuse-L* results, in which fragment components can be composed. This implies that a base language need not take precaution for genericity, extension, nor composition; instead, all necessary constructs are *derived* in the re-use language add-on and come for free. Since this principle is universal, grey-box component models for modeling and specification languages come for free, including attractive composition techniques, such as templates, semantic macros, views, role models, and aspects. And finally, using these principles, universal template expanders and aspect weavers can be built for all languages. Currently, our group works on such a generic toolset, *reuseware*, which can be downloaded from Sourceforge [Reuseware].

At the moment, UML is the main language for modeling in MDA. Thus, a grey-box UML component model seems to be indispensable for a fully generic and extensible MDA. Luckily, with add-on reuse languages, this component model should come for free, including UML template processors and weavers. Even, if in the future other languages are employed in the MDA stack, the universal technology will continue to work, so that on every stack level of the MDA re-use can be planned and unforeseen extensions can be provided for. This paves the way for true MDA components, both for commonality/variability and stability/extension scenarios.

References

[AOSD] The Aspect-Oriented System Development (AOSD) Community http://www.aosd.net.
[HyperJ] Ossher, H. et. al. Multi-Dimensional Separation of Concerns. The Hyperspace Approach. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm
[ISC] U. Aßmann. Invasive Software Composition. Springer, 2003.
[MDA] Model-Driven Architecture. OMG http://www.omg.org/mda
[Parnas] D. L. Parnas, On the Criteria to Be Used in Decomposing Systems into Modules,
Communications of the ACM, 15 (12), pp 1053-1058, Dec. 1972
[Reuseware] J. Johannes, J. Henriksson. The Reuseware toolkit. http://reuseware.sourceforge.net.
[REWERSE] Reasoning on the Web (REWERSE). EU Network of Excellence. Working Group I3 „Composition and Typing“. http://www.rewerse.net/i3
[Roles] T. Reenskaug. Role Modeling. http://heim.ifi.uio.no/~trygver

Certified Binaries for Software Components

Sagar Chaki, James Ivers, Kurt Wallnau
Software Engineering Institute

Introduction

There is an evident need for mechanisms that enhance our ability to *trust* third-party software. In the current era of plug-and-play, off-the-shelf programs are being increasingly made available as modules or *components* that can be attached to an existing infrastructure. More often than not, such plug-ins are distributed in machine code or *binary* form. In this article we present a framework that can be used to generate trustworthy binaries for software components, and to prove that binaries generated elsewhere satisfy specific policies. At the core of our methodology lies a paradigm called proof-carrying-code (PCC), originally proposed in a seminal paper by Necula and Lee [1, 2]. The essential idea underlying PCC is to construct a proof of the claim that a piece of machine code respects a desired *policy*. The proof is shipped along with the code so that it may be independently verified before the code is deployed.

To date, the application of PCC has been restricted to pure *safety* policies. The progress of PCC has also been hindered by, among other things, the need for manual intervention (e.g., discovering complicated loop invariants), and large proof sizes. Our approach overcomes these limitations of PCC in the context of certifying software components using powerful, but specific techniques. In particular, we achieve the following three objectives: (1) **Enrich:** Expand the set of PCC policies to include both safety and *liveness*. To this end, we use a state/event-based temporal logic called SE-LTL developed in the context of the Predictable Assembly from Certifiable Components (PACC)[1] project at the SEI. (2) **Automate:** Use iterative refinement in combination with predicate abstraction and model checking to generate appropriate invariants and ranking functions required for certificate and proof construction in a completely *automated* manner. (3) **Compact:** Use state-of-the-art Boolean satisfiability (SAT) technology to generate extremely small proofs. Preliminary investigations [5] indicate that the use of SAT yields proofs sizes that are several orders of magnitude more compact than when using conventional methods.

Background

In the original formulation of PCC, the world is divided into trusted code consumers and untrusted code producers. A code consumer publishes a safety policy. In general, safety policies assert that “something bad never happens,” while liveness policies assert “something good will eventually happen.” The code producer annotates the code with key invariants and uses a *certifying compiler* to generate object code as well as a verification condition (VC); in essence, the VC is the logical formula that is valid if and only if the object code respects the safety policy. The certifying compiler also constructs a proof of the VC, which is embedded in the object code; hence “proof carrying.” The code consumer checks that the proof is valid by verifying its construction against a set of sound axioms and inference rules that have been defined on the machine instructions themselves. The verification step is efficient, and reduces to a form of type checking. In other words, the proof is valid if, and only if, it is well-typed.

[1] http://www.sei.cmu.edu/pacc

PCC does not depend on the correctness of the certifying compiler or on the technologies used to construct proofs of program properties. PCC is also resilient to tampering, including code optimizations. Most attempts at modifying either the object code or the proof of the VC will lead to an ill-typed proof and hence will be detected. Moreover, any undetected tampering is guaranteed to result in code that still respects the published safety policy, and hence is harmless as far as the policy is concerned. Last, proof-carrying code is efficient, since the static proof eliminates the need for runtime checks. Still, a number of technical challenges (discussed in [2]) arose in this original formulation of PCC. Particularly notable among these are:

- **(CH1)** The restriction to safety conditions is problematic if the cost of developing a trustworthy PCC infrastructure is great.
- **(CH2)** The proof generator sometimes requires manual assistance, for example to compute loop invariants; for practical transition purposes, this is a non-starter.
- **(CH3)** The proofs generated are often quite large, hindering wider use of the PCC paradigm. Despite a lot of recent advances, this problem continues to be open.

Prior to this work, we conducted two projects that have a direct bearing on the above challenges. First, as part of the PACC project, we developed an expressive linear temporal logic called SE-LTL that can be used to express both safety and liveness claims of component-based software. In this work, we adopt and modify SE-LTL to express certifiable policies, thereby targeting **CH1**. Second, in collaboration with Prof. Peter Lee, an original proponent of and leading expert in PCC, we conducted an Independent Research and Development (IRAD) project [3] on "Assessing and Demonstrating the Readiness of Proof Carrying Code for Obtaining Objective Trust in Software Components". As part of this PCC-IRAD, we have developed an infrastructure to generate compact certificates for *C programs* (not binaries) against SE-LTL claims in an automated manner. The automation is achieved by combining iterative refinement with predicate abstraction and model checking to generate appropriate invariants and ranking functions that are required for certificate and proof construction. The tightness of proofs is obtained via the use of the state-of-the-art Boolean satisfiability (SAT) technology [5]. In this work, we extend this framework to certify *binaries* generated from component specifications. Thus, we complete the framework from a PCC perspective, and also address issues **CH2** and **CH3**. To this end we build on the PACC infrastructure for analyzing specifications of software component assemblies and generating deployable machine code for such assemblies.

Overall Approach

Our technical approach is best summarized by the architecture described in Figure 1. This figure depicts the final infrastructure for certified component binary generation that we developed. The boxes are numbered for ease of reference. The steps involved in generating certified component binaries can be summarized as follows:

1. We begin (box 1) with a specification of a component assembly written in the Construction and Composition Language (CCL) [9], which has been developed as part of the PACC project. A CCL specification contains a description of the assembly as well as safety and liveness policies that need to be certified. CCL is currently implemented as a profile of an executable subset of UML 2.0.
2. The CCL specification is automatically *interpreted* [4] into a form that can be processed by a model checker. This form (box 2) essentially comprises of a C

program along with finite state machine specifications for library routines invoked by the program. The interpretation procedure was implemented as part of the PACC project.

3. The result of the interpretation is input to *Copper* (box 3), a state-of-the-art *certifying software model checker*. Copper [8] was originally developed as part of the ComFoRT [7] reasoning framework of the PACC project. It was enhanced [5] with the ability to generate certificates and proofs as part of the PCC-IRAD [3]. Copper interfaces with theorem provers (TP) and SAT solvers (SAT) during model checking and certificate generation. The output of Copper is either a counterexample to the policy (CE) or a proof (Proof1) that the input to Copper respects the desired policies.
4. Proof1 only certifies that the result of interpreting the original CCL specification respects the desired policies. It is *reverse-interpreted* (arrow 4) into a proof (Proof2) that the CCL specification itself also respects these policies. However, in order to generate certified binaries, we perform two additional steps.
5. The CCL specification is now transformed (arrow 5) into a Pin/C program (box 6) that can be compiled and deployed in a Pin runtime environment (RTE). This transformation process, as well as the Pin RTE, has been developed to a large extent as part of the PACC project. We enhanced this transformation process so that it also creates a proof of the correctness of the generated Pin/C code from the proof of the correctness of the CCL specification. In essence, we transform the proof of correctness, along with the actual assembly, from one format (CCL) to another (Pin/C).
6. The final step (arrow 7) is conceptually the same as the previous step. We use a standard C compiler (gcc) to achieve this goal. The end result is a proof (Proof3) that the final (i.e., binary code for the) component assembly respects the desired policies.

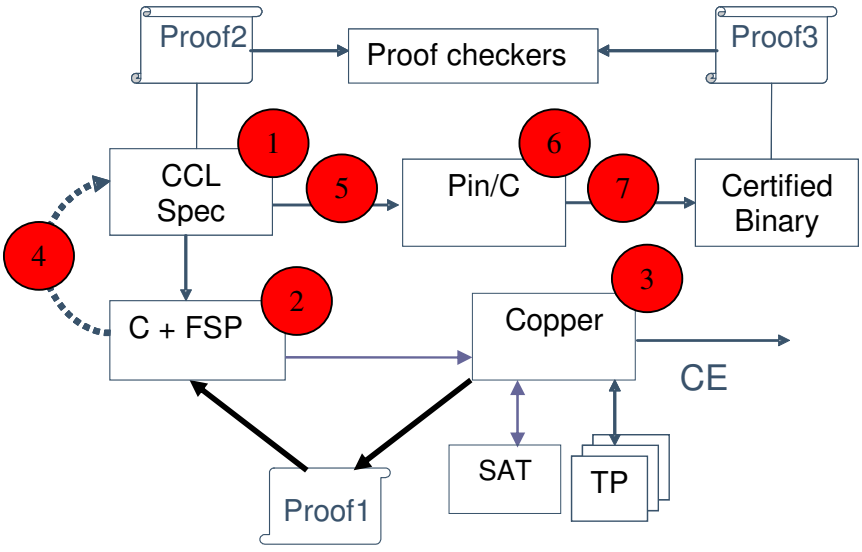

Figure 1: Architecture of developed framework.

References

[1] George Necula, "Proof-carrying code," In Proc. 24th ACM Symposium on Principles of Programming Languages (POPL), New York, Jan. 1997 (106-119).

[2] George Necula and Peter Lee, "Safe kernel extensions without runtime checking," In Proc. 2nd USENIX Symposium on Operating System Design and Implementation (OSDI), Seattle, Wshington1996 (229-243).

[3] Sagar Chaki, Kurt Wallnau, "Proof-Carrying Code," CMU/SEI-2005-TR-020, Chapter 6, December 2005.

[4] James Ivers, Nishant Sinha and Kurt Wallnau, "A Basis for Composition Language CL", (CMU/SEI-2002-TN-026).

[5] Sagar Chaki, "SAT-based Software Certification", in Proc. Of TACAS, 2006.

[6] Fred Schneider, "Enforceable Security Properties," in ACM Transactions on Information and System Security, Vol. 3, No. 1, February 2000 (30-50).

[7] James Ivers and Natasha Sharygina, "Overview of ComFoRT: A Model Checking Reasoning Framework", (CMU/SEI-2004-TN-018).

[8] Sagar Chaki, James Ivers, Natasha Sharygina and Kurt Wallnau, "The ComFoRT Reasoning Framework," in Proc. 17th Computer Aided Verification (CAV), LNCS 3576 (164-169), July 2005.

[9] Kurt Wallnau and James Ivers, "Snapshot of CCL: A Language for Predictable Assembly", (CMU/SEI-2003-TN-025).

"De klant wil snel op de markt zijn met de nieuwste producten. Daar zorg ik voor en dat geeft een geweldige kick!"

UMTS, GPRS, Bluetooth... Hoe kies jij de slimste stack?

Keuzes weeg je af. Zeker als het gaat om een tweede carrièrestap. Jij wilt verder: meer technische uitdagingen en meer ruimte voor persoonlijke groei. De keuze is dan eenvoudig. Philips TASS, voor een carrière in de technische software ontwikkeling.

Ga snel naar www.benjijeentasser.nl

TASS software professionals

Agile Model Driven Development (AMDD)

Scott W. Ambler
Practice Leader Agile Development, IBM

Modeling is an important part of all software development projects because it enables you to think through complex issues before you attempt to address them via code. This is true for agile projects, for not-so-agile projects, for embedded projects, and for business application projects. Unfortunately, many modeling efforts prove to be dysfunctional. At one end of the spectrum are projects where no modeling is performed, either because the developers haven't any modeling skills or because they have abandoned modeling as a useless endeavor. At the other end of the spectrum are projects which sink in a morass of documentation and overly detailed models, either because the project team suffers from "analysis paralysis" and finds itself unable to move forward or because the team has burdened itself with too many modeling specialists who don't have the skills to move forward even if they wanted to. Somewhere in the middle are project teams that invest in modeling and documentation efforts only to discover that the programmers ignore the models anyway, often because the models are unrealistic or simply because the programmers think they know better than the modelers (and often they do). The goal of Agile Model Driven Development (AMDD) is to show how to avoid these problems, to gain the benefits of modeling and documentation without suffering the drawbacks.

1 Agile Models

A *model* is an abstraction that describes one or more aspects of a problem or a potential solution addressing a problem. Traditionally, models are thought of as zero or more diagrams plus any corresponding documentation. However non-visual artifacts such as use cases, a textual description of one or more business rules, or a collection of class responsibility collaborator (CRC) cards [1] are also models. An *agile model* [2] is a model that is just barely good enough [18] for the situation at hand. Agile models are just barely good enough when they exhibit the following traits:

- Agile models fulfill their purpose.
- Agile models are understandable.
- Agile models are sufficiently accurate.
- Agile models are sufficiently consistent.
- Agile models are sufficiently detailed.
- Agile models provide positive value.
- Agile models are as simple as possible.

Figures 1 and 2 both depict agile models. Figure 1 depicts a hand-drawn architecture sketch for a business application which was created by the team on the first few days of the project. Figure 2 depicts a physical data model (PDM) using the Unified Modeling Language (UML) notation [3] – it is possible to data model effectively using the UML. Both models are agile even though they're very different from each other:

- The data model is very likely a keeper whereas the sketch would be discarded once it's served its purpose.
- The data model was created using a sophisticated modeling tool whereas the sketch was created using very simple tool.
- The data model was created using a sophisticated notation, yet the sketch is clearly free-form.

- The data model depicts technical, detailed design whereas the sketch is high-level.

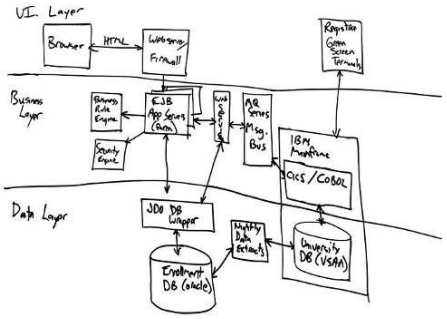

Figure 1: A hand-drawn architectural sketch.

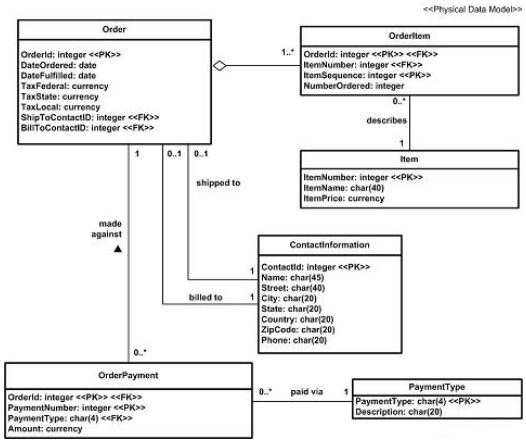

Figure 2: A physical data model (PDM).

One of the more controversial concepts in Agile Modeling is that agile models and agile documents are sufficient for the task at hand, or as I like to say they are "just barely good enough". For some reason people think that just barely good enough implies that the artifact isn't very good, when in fact nothing could be further from the truth. When you stop and think about it, if an artifact is just barely good enough then by definition it is at the most effective point that it could possibly be at. Figure 3 summarizes the value curve for an artifact being just barely good enough. Value refers to the net benefit of the artifact, which would be calculated as benefit - cost. The dashed line is at the point where the artifact is just barely good enough: anything to the left of the line implies that you still have work to do, anything to the right implies that you've done too much work. When you are working on something and it isn't yet barely good enough then you can still invest more effort in it and gain benefit from doing so (assuming of course you actually do work that brings the artifact closer to it's intended purpose). However, if an artifact is already just barely good enough (or better) then doing more work on it is clearly a waste: once an artifact fulfils its intended purpose then any more investment in it is useless bureaucracy. The diagram is a little naive because it is clearly possible for the value to be negative before the artifact becomes barely good enough although for the sake of argument I'm going to assume that you do a good job right from the beginning.

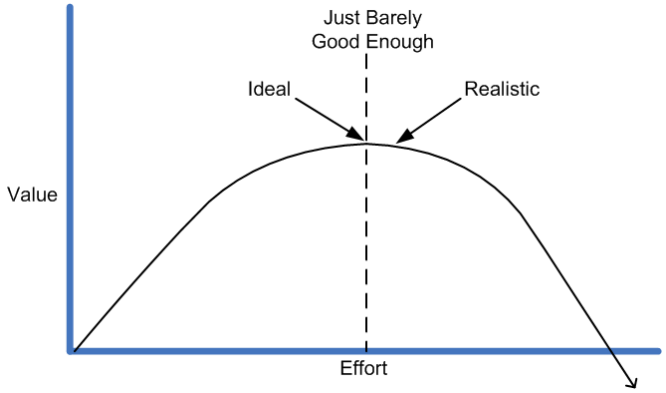

Figure 3: Just barely good enough is the most efficient.

The fundamental challenge with "just barely good enough" is that it is situational. For example, I often draw UML Sequence diagrams on a whiteboard to explore complex logic and then discard it once I'm done with it. In this case the whiteboard diagram is fine because it helps me to solve the issue which I'm thinking through with whomever I'm working. But, what if we're in a situation where we'll need to update this logic later on AND will want to do it via the diagram instead of via source code? Clearly a hand-drawn sketch isn't good enough in this case and we'll want to create a detailed diagram using a sophisticated CASE tool. We'd still create an agile model even though it is much more sophisticated than a sketch because "just barely good enough" reflects the needs of the situation.

It is important to distinguish between the orthogonal concepts of models and documents: some models become documents, or parts of documents, although many models are discarded after they have been used. I suspect that 90% or more of all models are discarded – how many whiteboard sketches have you erased throughout your career? For the sake of definition a document is a permanent record of information, and an *agile document* [2] is a document that is just barely good enough. The principles and practices of Agile Modeling, described in the next section, are applicable to both modeling and documentation.

2. Agile Modeling (AM)

The Agile Modeling (AM) method defines a collection of values, principles, and practices which describe how to streamline your modeling and documentation efforts. These practices can be used to extend agile processes such as Extreme Programming (XP) [4], Feature Driven Development (FDD) [5], and Rational Unified Process (RUP) [6]. AM is a chaordic collection of practices – guided by principles and values – that should be applied by software professionals on a day-to-day basis. The focus of AM is to make your modeling and documentation efforts lean and effective; AM does not address the complete system lifecycle and thus should be characterized as a partial process/process. The advantage of this approach is that organizations may benefit from the focused guidance of a partial process. The disadvantages are that organizations need the requisite knowledge and skills to know which processes exist and how to combine them effectively. The concept of partial processes seems strange at first, but when you reflect a bit you quickly realize that partial processes are the norm – development processes, such as XP and the RUP, address the system development lifecycle but do not address the full IT lifecycle. The Enterprise Unified Process (EUP) [7] – an extension to the RUP which addresses the production and retirement phases of a system, operations and support

of a system, and cross-system issues such as enterprise architecture and strategic reuse – represents a full IT lifecycle.

AM is practices-based, it is not prescriptive. In other words it does not define detailed procedures for how to create a given type of model, instead it provides advice for how to be effective as a modeler. The advantage of describing a process as a collection of practices is that it is easy for experienced professionals to learn and reflects (hopefully) what they actually do, the disadvantage is that it does not provide the detailed guidance for novices. Prescriptive processes, on the other hand, often provide the detailed guidance required by novices but are ignored by experienced professionals. Prescriptive processes are well suited as training material for new hires and perhaps as input into process audits to fulfill the requirement that you have a well documented process.

Think of AM as more of an art than a science. It is defined as a collection of values, principles, and practices. (www.agilemodeling.com/values.htm), (www.agilemodeling.com/principles.htm), (www.agilemodeling.com/practices.htm). The values of AM include those of XP v1 [20]– *communication*, *simplicity*, *feedback*, and *courage* – and extend it with *humility* (XP v2 [21] adds the fifth value of *respect*, which I argue comes from *humility*). The principles of AM, many of which are adopted or modified from XP, provide guidance to agile developers who wish to be effective at modeling and documentation. They provide a philosophical foundation from which AM's practices are derived. The practices of AM are what people actually do. There is not a specific ordering to the practices, nor are there detailed steps to complete each one – you simply do the right thing at the right time.

Because every project team is different, and every environment is different, you should tailor your process to reflect your situation. AM reflects this philosophy – to claim that you are "doing AM" you merely need to adopt its values, its core principles and practices (see Table 1). The remaining principles and practices are optional, although they are very good ideas and should be adopted whenever possible. This approach enables you to tailor AM to meet your exact needs. Table 2 lists the supplementary principles and practices although for brevity does not describe them in detail.

Why would you want to adopt AM? AM defines and shows how to take a light-weight approach to modeling and documentation. What makes AM a catalyst for improvement is not the modeling techniques themselves – such as use case models, class models, data models, or user interface models – but how to apply them productively. Although you must be following an agile software process to truly be agile modeling, you may still adopt and benefit from many of AM's practices on non-agile projects.

Table 1. The core principles and practices of AM.

Core Principles	Core Practices
• Assume Simplicity • Embrace Change • Enabling the Next Effort is Your Secondary Goal • Incremental Change • Maximize Stakeholder Investment • Model With a Purpose • Multiple Models • Quality Work • Rapid Feedback • Software is Your Primary Goal • Travel Light	• Active Stakeholder Participation • Apply the Right Artifact(s) • Collective Ownership • Single Source Information • Create Several Models in Parallel • Create Simple Content • Depict Models Simply • Display Models Publicly • Iterate To Another Artifact • Model in Small Increments • Model With Others • Prove it With Code • Use the Simplest Tools

Table 2. Supplementary principles and practices.

Supplementary Principles	Supplementary Practices
• Content is more important than representation • Open and honest communication • Work with people's instincts	• Apply modeling standards • Apply patterns gently • Discard temporary models • Formalize contract models • Update only when it hurts

3. Agile Model Driven Development (AMDD)

As the name implies, AMDD is the agile version of Model Driven Development (MDD). MDD is an approach to software development where extensive models are created before source code is written. A primary driver of MDD is the Object Management Group (OMG)'s Model Driven Architecture (MDA) standard [11]. With MDD the goal is typically to create comprehensive models, and then ideally generate software from those models. This is a great vision, but one that may not be possible for all development teams.

AMDD takes a much more realistic approach: its goal is to describe how developers and stakeholders can work together cooperatively to create models which are just barely good enough. It assumes that each individual has some modeling skills, or at least some domain knowledge, that they will apply together in a team in order to get the job done. It is reasonable to assume that developers will understand a handful of the modeling techniques out there, but not all of them. It is also reasonable to assume that people are willing to learn new techniques over time, often by working with someone else that already has those skills. AMDD does not require everyone to be a modeling expert, it just requires them to be willing to try. AMDD also allows people to use the most appropriate modeling tool for the job, often very simple tools such as whiteboards or paper, because you want to find ways to communicate effectively, not document comprehensively. There is nothing wrong with sophisticated CASE tools in the hands of people who know how to use them, but AMDD does not depend on such tools.

Figure 4 depicts a high-level lifecycle for AMDD for the release of a system [9]. Each box represents a development activity. The initial up front modeling activity occurs during cycle/iteration 0 and includes two main sub-activities, initial requirements modeling and initial architecture modeling. The other activities – model storming, reviews, and implementation – potentially occur during any cycle, including cycle 0. The time indicated in each box represents the length of an average session: perhaps you will model for a few minutes then code for several hours.

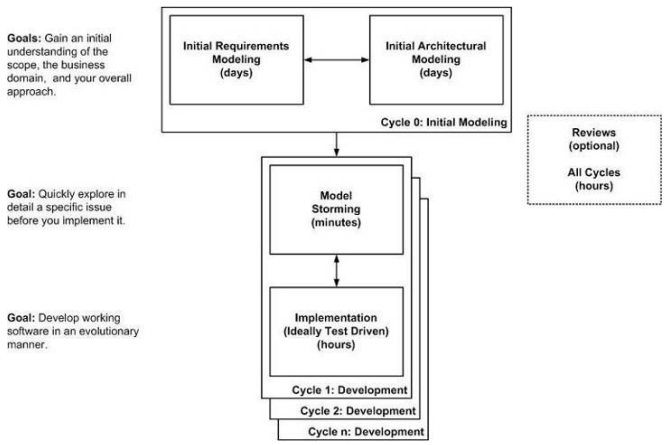

Figure 4. Taking an AMDD approach to development.

3.1 Initial Modeling

The initial modeling effort is typically performed during the first week of a long-term project. For short projects (perhaps several weeks in length) you may do this work in the first few hours and for longer projects (perhaps on the order of twelve or more months) you may decide to invest up to two weeks in this effort. You should not invest any more time than this as you run the danger of over modeling and of modeling something that contains too many problems (two weeks without the concrete feedback that implementation provides is a long time to go at risk).

Initial modeling occurs during cycle 0, the only time that an agile modeler will spend more than an hour or two at once modeling because they follow the practice *Model in Small Increments*. During cycle 0 you are likely to identify high-level usage requirements models such as a collection of use cases or scenarios; identify high-priority technical requirements and constraints; create a high-level (sparse) domain model; and draw sketches representing critical architectural aspects of your system. In later cycles both your initial requirements and your initial architectural models will need to evolve as you learn more, but for now the goal is to get something that is just barely good enough so that your team can get coding. In subsequent releases you may decide to shorten cycle 0 to several days, several hours, or even remove it completely as your situation dictates.

3.2 Model Storming

During development cycles you explore the requirements or design in greater detail, and your "model storming" sessions are often on the order of minutes. Model storming is a just-in-time (JIT) approach to modeling with a twist – you model just in time and just enough to address the issue at hand. Perhaps you will get together with a stakeholder to analyze the requirement you're currently working on, create a sketch together at a whiteboard for a few minutes, and then go back to coding. Or perhaps you and several other developers will sketch out an approach to implement a requirement, once again spending several minutes doing so. Or perhaps you and your programming pair will use a modeling tool to model in detail and then generate the code for that requirement. Model storming sessions shouldn't take more than 15 or 20 minutes, otherwise you're likely not following the AM practice *Iterate to Another Artifact* properly, and often take a few minutes at most.

It's important to understand that your initial requirements and architecture models will evolve through your detailed modeling and implementation efforts. That's perfectly natural. Depending on how light you're traveling, you may not even update the models if you kept them at all.

You may optionally choose to hold model reviews and even code inspections, but these quality assurance (QA) techniques really do seem to be obsolete with agile software development. Although many traditionalists consider model reviews to be best practices they're really "compensatory practices" that compensate for common process-oriented mistakes such as:

- Distributing your team across several locations, thereby putting you at risk that the teams are not aware of what the others are doing.
- For allowing one person or a subset of people (often specialists) to "own" the model, thereby putting you at risk that the model is of poor quality or does not reflect the work of the others on the team.
- For long feedback loops, such as a (near) serial approach to development when it can be months or even years between modeling and coding activities.

When you follow AM's practices of *Active Stakeholder Participation*, *Collective Ownership*, *Model With Others*, and *Prove it With Code* you typically avoid these problems. The high-communication and open environment enjoyed by agile modelers ensures that many people, if not everyone on the team, works with all artifacts. This ensures that many "sets of eyes" see any given model, thereby increasing the chance that mistakes are found early. The focus on producing working software ensures that the ideas captured in models are quickly put to the test – very often something will be modeled and then implemented the very same day. In these environments the value of reviews quickly disappears.

3.3. Implementation

Implementation is where your team will spend the majority of its time. During development it is quite common to model storm for several minutes and then code, following common agile implementation practices for several hours or even days. These implementation practices are:

1. **Code refactoring**. Refactoring [12] is a disciplined way to restructure code to improve its design. A code refactoring is a simple change to your code that improves its design but does not change its behavioral semantics.
2. **Database refactoring**. A database refactoring [10] is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. There are different types of database refactorings. Some focus on data quality (such applying a consistent format to the values stored in a column), some focus on structural changes (such as renaming or splitting a column), whereas others focus on performance enhancements (such as introducing an index). Structural database refactorings are the most challenging because a change to the structure of your database could cause your application (or others) to crash.
3. **Test-Driven Development (TDD)**. Test-driven development (TDD) [13, 14], also known as test-first programming or test-first development, is an approach where you identify and write your tests before your write your code. There are four basic steps to TDD. First, you quickly add a test (just enough code to fail), the idea being that you should refuse to write new code unless there is a test that fails without it. The second step is to run your tests, either all or a portion of them, to see the new test fail. Third, you make a little change to

your code, just barely enough to make your code pass the tests. Next you run the tests and hopefully see them all succeed – if not you need to repeat step 3. There are several advantages of TDD. First, it ensures that you always have a 100% unit regression test suite in place, showing that your software actually works. Second, TDD enables you to refactor your code safely because you know you can find anything that you "break" via a refactoring. Third, TDD provides a way to think through detailed design issues, reducing your need for detailed modeling.

These three techniques are effectively enablers of AMDD. Refactoring helps you to maintain a quality design within your object schema over time and supports detailed changes to your design that aren't captured within your design models. Similarly database refactoring helps you to maintain a quality design within your data schema, in many ways it could be thought of as normalization after the fact. Both techniques push evolutionary design decisions into the hands of the people most qualified to make them – the people actually building the system. AMDD and TDD go hand-in-hand because they are both "think before you code" techniques. AMMD provides a way to think through big issues whereas TDD provides a way to think through detailed issues.

4. Conclusion

Modeling is a skill that all developers must gain to be effective. Agile Modeling (AM) defines a collection of values, principles, and practices which describe how to streamline your modeling and documentation efforts. Modeling can easily become an effective and high-value activity if you choose to make it so; unfortunately many organizations choose to make it a bureaucratic and documentation-centric activity which most developers find intolerable.

The Agile Model Driven Development (AMDD) process describes an approach for applying AM in conjunction with agile implementation techniques such as Test Driven Development (TDD), code refactoring, and database refactoring. AMDD enables agile developers to think through larger issues before they dive down into the implementation details. AMDD is a valuable technique to have in your intellectual toolbox.

5. Resources

1. Beck, K., and Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. In *Proceedings of OOPSLA'89*, pp. 1–6.
2. Ambler, S.W. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons Publishing. 2002
3. Ambler, S.W. *An Unofficial Profile for Data Modeling Using the UML*. www.agiledata.org/essays/umlDataModelingProfile.html
4. Beck, K. *Extreme Programming Explained – Embrace Change*. Reading, MA: Addison Wesley Longman, Inc. 2000
5. Palmer, S. R. & Felsing, J. M. *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ: Prentice Hall PTR. 2002.
6. Kruchten, P. *The Rational Unified Process 2nd Edition: An Introduction*. Reading, MA: Addison Wesley Longman, Inc. 2000
7. Ambler, S.W., Nalbone, J, and Vizdos, M.J. *The Enterprise Unified Process: Extending the Rational Unified Process*. Upper Saddle River, NJ: Prentice Hall PTR. 2005.
8. Cockburn, A. *Agile Software Development*. Reading, MA: Addison Wesley Longman, Inc. 2002
9. Ambler, S.W. *The Object Primer 3rd Edition: Agile Model Driven Development with UML 2*. New York: Cambridge University Press, 2004.

10. Ambler, S. W. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. New York: Wiley. 2003.
11. *Model Driven Architecture (MDA) Home Page*. www.omg.org/mda/
12. Fowler, M. *Refactoring: Improving the Design of Existing Code*. Menlo Park, CA: Addison Wesley Longman. 1999.
13. Astels, D. *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall. 2003.
14. Beck, K. *Test Driven Development: By Example*. Boston, MA: Addison Wesley. 2003.
15. Ambler, S.W. *Agile Model Driven Development is Good Enough*. IEEE Software, September/October 2003, 20(5), pp. 70-73.
16. Ambler, S.W. *Inclusive Modeling*. www.agilemodeling.com/essays/inclusiveModeling.htm. 2004.
17. Breen, P. *Software Craftsmanship*. Boston, MA: Addison Wesley. 2002.
18. Ambler, S.W. Just Barely Good Enough Models and Documentation? www.agilemodeling.com/essays/barelyGoodEnough.html
19. Ambler, S.W. *The Elements of UML 2.0 Style*. New York: Cambridge University Press. 2005.
20. Beck, K. *eXtreme Programming eXplained: Embrace Change.* Addison Wesley. 1999
21. Beck, K. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison Wesley. 2004

Bio:

Scott W. Ambler (www-306.ibm.com/software/rational/bios/ambler.html) is Practice Leader Agile Development with IBM's methods group and a Senior Contributing Editor with Dr. Dobb's Journal. He is founder and thought leader of the Agile Modeling (AM) (www.agilemodeling.com), Agile Data (AD) (www.agiledata.org), and Enterprise Unified Process (EUP) (www.enterpriseunifiedprocess.com) methodologies. He is (co-)author of 19 IT-related books, several of which have won industry awards.

Océ R&D: voor Embedded Software Specialisten die meer afwisseling zoeken.

Als Embedded Software Specialist kun je bijna overal aan de slag. Maar als je liever niet te maken krijgt met de beruchte hokjes-geest, kom je al gauw bij Océ R&D. Want bij ons werk je in multidisciplinaire teams. Samen met onder meer elektrotechnici en werktuigbouwers, chemici en fysici. Daarbij wordt van jou verwacht dat je steeds snel complexe problemen analyseert en creatieve oplossingen bedenkt.

Zoals je misschien weet is Océ een van ’s werelds grootste printerfabrikanten. In een sterk concurrerende markt blijft Océ succesvol dankzij eigen technologie, focus op klanten en slimme innovaties. Software maakt daarbij steeds meer het verschil.

Als R&D-teamlid werk je in een informele cultuur, waar je het beste uit jezelf kunt halen. En Océ biedt talloze doorgroeimogelijkheden in techniek, management en business. Zowel in Nederland als daarbuiten. Uiteraard verwachten we veel van onze nieuwe embedded software specialisten. Je hebt een relevante universitaire studie (bijna) voltooid. En als persoon ben je analytisch, oplossingsgericht, creatief én initiatiefrijk. Sta je aan het begin van je loopbaan, neem nu het initiatief en maak kennis met de R&D van Océ. Surf naar onze site jobs.oce.com of bel 077 359 3011 voor meer informatie of een afspraak.

océ

Printing for Professionals

Aspect Orientation Enables Differentiation with Software

Piërre van de Laar (pierre.van.de.laar@esi.nl)
TU/e Campus, Laplace-building 0.10
Den Dolech 2
P.O.Box 513, 5600 MB Eindhoven
The Netherlands

Companies develop large numbers of embedded systems that contain a significant amount of software. The amount of software in these embedded systems is exponentially growing according to Moore's law. With the increase of software also the complexity increases. Separation of concerns, i.e., the ability to deal with the difficulties, the obligations, the desires, and the constraints one by one [Dijkstra, 1976], is needed to cope with this growing complexity and to ensure that companies can continue to differentiate with software.

Currently, embedded software is modularized based on functionality. Unfortunately, this kind of modularization cannot separate all concerns as can be observed in the current software:

- Many (non-functional) concerns are not localised in one software module but are scattered throughout the software.
- Multiple concerns are tangled in one software module.

Since concerns are not separated, complexity increases; independence decreases; and decision points must be preponed. Throughout the whole software development process, the impact of a limited separation of concerns is noticeable:

- Traceability and localisation of requirements is reduced.
- Independent (multi-site) development and the associated integration and validation is prevented.
- During implementation, code is duplicated which not only is the root cause of errors due to inconsistencies, but also prevents specialisation and wastes scarce and limited [Kaashoek, 2005] developer resources.
- Maintenance becomes more difficult.
- Evolution and reuse within a product family is hampered.

To solve these problems, we ask ourselves the question: how can we improve the effectiveness of our modularization?

Aspect orientation is a technology that improves the effectiveness of modularization. Aspect orientation modularizes the system based on concerns. In 1978 [Sandewall, 1978] the principles of aspect orientation were described for the first time. Yet, it became hot due to Xerox [Kiczales *et al.*, 1997], who applied aspect orientation amongst others for image improvements. Currently, on top of every popular programming language an aspect oriented programming language exists. For example,

- AspectC++,
- AspectC,

- AspectSharp, and
- AspectJ.

The interest for aspect orientation is not limited to these Open Source projects, also Microsoft is investigating it for their Developer Studio[2]. We have investigated aspect orientation in the scope of hybrid (analog and digital) television.

In the remainder of this extended abstract, we will first give a black and white picture of aspect orientation. For a more in depth description, we recommend [Kiczales et al., 1997, Elrad et al., 2001, Laddad, 2003, Filman et al., 2005] to the interested reader. We will then describe how we added aspects to the component-based software of television, and share our experiences with applying aspect orientation in this context. We will end with a summary.

What is aspect orientation?

Aspect orientation introduces join points around the execution of instructions to handle concerns related to these instructions. Join points are also the only points where aspects can interact with other pieces of software. To give an example, join points around instructions that change items in a database for a user, enable that:

- Before the instructions are executed, the access to the database is logged;
- The instructions are only executed when the user has the rights to modify the items in the database; and
- After execution of the instructions, all observers of the database are notified to ensure accurate visualizations.

An aspect contains pointcuts and advices. A pointcut specifies, by selecting join points, where an aspect crosscuts other aspects. Join points can be selected based on, amongst others, the type and name of functions and its parameters. An advice specifies in a function-like construct what behaviour to exhibit around the selected join points. For the implementation of an advice, an aspect may require functionality of other aspects, use meta-data about the selected join point, and introduce variables.

Making a product from aspects is called weaving: joining the aspects at the selected join points. Weaving can occur at different points in time. To give a few examples: Before compile time by code weaving, at load-time by the class loader, or at run-time by the virtual machine.

Adding aspect orientation to component-based software

Component-based software has besides source and binary/byte code also an architectural description. We decided to weave based on these architectural descriptions to leverage the following advantages:

1. The architectural description contains information, some of which is lost in the source code. For example, since the C programming language has no interface concept, the information of which functions constitute an interface is lost. Similarly, the direction of parameters of functions is lost in C.
2. The source code of a component is often not available, while the architectural description is always available. But even when source code is available,

[2] For more info, see http://research.microsoft.com/workshops/aop/.

weaving at source code level typically invalidates the warranty and support of components.
3. The architectural description language is implementation-language agnostic, which makes the weaving implementation-independent.
4. The sensitivity of the system for modifications at architecture level is by design less than at the source code level. Computations that cross component boundaries must be able to handle the allowed variations in the implementation of interfaces and are thus less sensitive for modification compared to computations within a component that typically exploit implementation details to optimize throughput and response time.
5. The architectural description has a higher abstraction level and is more stable than the implementation; this positively influences the independent evolution of aspects and components.

Which components can be affected by an aspect? Even though the composition of a component is implementation dependent, we decided that an aspect could affect all components in a product. This choice enables more powerful aspects, which are needed, amongst others, for logging all components, and asserting that all components are only used after initialisation.

What are the join points in an architectural description? We consider the functions in the interface of a component as join points, since:

- A component only communicates via these functions.
- Developers explicitly describe both the functions in an interface and the interfaces of a component.
- Only these functions are implementation-independent.

Experiences with aspect orientation

We first gained experience with and confidence in aspect orientation in the validation and verification phase. This path ensured that we reduced the risks associated with our ultimate goal: The introduction of aspect orientation into our products.

How to handle access before initialisation is a concern that affects all components. Although one can easily describe how to handle access before initialisation in general, it is currently handled per component. Even worse, this handling differs between components in the same software stack. With aspect orientation, we were able to write three different strategies to handle access before initialisation. The first strategy asserts that a component is not accessed before initialisation; the second strategy ignores accesses when the component is not yet initialised; and the third strategy calls the initialisation code when the component is accessed but not initialised before. With these strategies:

1. We could ensure that all components handle access before initialisation identically.
2. We could separate the initialisation implementation from the functional implementation. This not only reduces the lines of code by 2%, but also makes reuse more likely. Reuse becomes more likely since the reuse environment has only to match either the initialisation requirement (to reuse one of the three initialisation aspects) or the functional requirement (to reuse one of the components), but not both.
3. We could postpone the decision for an initialisation strategy from implementation to integration.

Resource usage is an important concern for resource limited systems. The functionality to check that a component does not use more resources than specified can be localised in one component. Yet, for each component under test one still has to do a lot of plumbing:

- Instantiate a test component, and
- Change the connections to the component under test to pass through this test component.

With aspect orientation, we were able to localise not only the functionality but also the plumbing in one aspect. This made the test process both easier and less error-prone.

Many pieces of software cannot be accessed multithreaded, but accidentally are accessed on multiple threads. Integration and testing would benefit from automatic detection of illegal multithreaded accesses. We have written an aspect that lists multithreaded accesses throughout the complete software stack. This list can help architects to pinpoint illegal multithreaded accesses. Of course, by adding attributes to the current code base and exploiting this information in a comparable aspect, also this last step can be automated [Hoogendijk *et al.*, 2005].

During integration and testing, understanding the dynamic behaviour is crucial. Tracing provides insight in this behaviour. We have written and applied an aspect to trace the interface function calls in an already finished television set. While manually adding trace statements requires programming effort linear with the number of interface function calls, this aspect required only a small programming effort that is independent of the number of interface function calls. Currently, we are using this aspect at NXP, the former Philips Semiconductors, to determine how the platform is accessed by the applications running on top of it.

Summary

Aspect orientation improves our effectiveness to modularize software. As a consequence, separation of concerns is better supported. This reduces the complexity of the software, minimizes dependencies, prevents duplication, ensures consistency, makes reuse more likely, and enables to postpone design decisions. Our experience indicates that aspect orientation scales to industrial applications. Furthermore, with aspect orientation, companies will be able to continue the differentiation with software.

References

[Dijkstra, 1976] Edsger W. Dijkstra, 1976, *A Discipline of Programming*, ISBN 0613924118.

[Sandewall, 1978] Erik Sandewall, 1978, *Programming in an Interactive Environment: the "LISP" Experience*, Computing Surveys, Vol. 10, No. 1, pp. 35-71.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, *Aspect-Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming, pp. 220-242. Available at http://www.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/for-web.pdf

[Elrad *et al.*, 2001] Elrad, T., Filman, R.E., Bader, A., October 2001. Special section on Aspect-Oriented Programming.Communications of the ACM, Vol. 44 No. 10, pp. 29-97.

[Laddad, 2003] Laddad, R., 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning, ISBN 1-930-11093-6.

[Filman *et al.*, 2005] Filman, R.E., Elrad, T., Clarke, S., Aksit, M., 2005. *Aspect-Oriented Software Development*, Addison-Wesley, ISBN 0-321-21976-7.

[Hoogendijk *et al.*, 2005] Paul Hoogendijk, Chritiene Aarts, Piërre van de Laar, Felix Ogg, Rob van Ommering, Jur Pauw, *Extending the Nexperia Home Component Model: Annotating and Checking Thread-safety Properties*, Philips Software Conference 2005.