Contact Information

**Programming Technology Lab** *(Prof. Dr. Theo D’Hondt, Prof. Dr. Wolfgang De Meuter)*

Vrije Universiteit Brussel, DINF-PROG, Pleinlaan 2, B-1050 Brussels, Belgium

http://prog.vub.ac.be/

**System and Software Engineering Lab** *(Prof. Dr. Viviane Jonckers)*

Vrije Universiteit Brussel, INFO-ssel, Pleinlaan 2, B-1050 Brussels, Belgium

http://ssel.vub.ac.be/

This brochure was created in the context of the Interuniversity Attraction Poles Programme - Belgian State - Belgian Science Policy

http://www.belspo.be/
Research Descriptions 2006-2007

System and Software Engineering & Programming Technology Labs

Prof. Dr. Theo D'Hondt
Prof. Dr. Viviane Jonckers
Prof. Dr. Wolfgang De Meuter
Concept-Centric Coding for Software Evolution in an Agile Context

Concept-Centric Coding (C3) aims at providing programmer support to cope with software evolution in an agile context. It prescribes to complement an application’s source code with explicit knowledge about the application and its problem domain. This domain knowledge is described in a concept graph, which is more than a mere ontology. In addition to the domain concepts, it contains the source code entities of the application itself, thereby connecting the application’s domain knowledge with its implementation, and keeping this link up to date when the application evolves. Moreover, the domain knowledge actively contributes to the runtime functionality of the application. Hence, it becomes possible to adapt an applications’ behavior by modifying the corresponding domain concepts.

The 'Concept to Code Browser' (CoBro), is the core tool suite supporting the C3 approach. It is closely integrated with Smalltalk and the VisualWorks development environment. Moreover, it is implemented in symbiosis with Smalltalk allowing a programmer to transparently invoke and manipulate concepts as if they were plain Smalltalk objects. This enables a non-obtrusive synergy between the concept level and the code level. Part of the power of CoBro is realised by its (partial) metacircular implementation. In particular, CoBro itself is implemented in terms of domain concepts using the C3 approach. Consequently, it can be adapted, even at runtime, by changing its concepts, thus leading to a highly extensible and flexible tool-base.

dirk.deridder@vub.ac.be

Software evolution, domain modelling, flexible architectures, meta programming, software variability, co-evolution

BEHAVE - Verifying and Documenting Design Invariants in Software

Today’s software systems have to evolve rapidly, especially because of the quickly evolving business requirements. Supporting this process of change has up until now mainly resulted in tool support for modularizing systems better and localizing those parts sensitive to change.

But what about those parts of a software system that are not allowed to change when software evolves? In our approach, we offer support for documenting and verifying design invariants of a system. Design invariants represent the laws of your software system that must continue to hold through every evolution cycle. They are difficult to capture as they cross-cut an entire system. As an example of an invariant, consider a banking application: it should always be the case that, if a transaction error occurs, the outcome is in favor of the customer.

BEHAVE offers a platform to specify and verify a high-level behavioral model representing such a design invariant. The main contribution of our approach is that the behavioral models are specified using a declarative formalism which renders the models machine-verifiable but also understandable to developers. More specifically, BEHAVE lets you specify a behavioral model of the invariant over selective high-level run-time events using temporal logic programming.

isabel.michiels@vub.ac.be

Dynamic analysis, declarative meta programming, lightweight verification
In order to survive in today's highly dynamic marketplace, companies must show a continuous and ever-increasing ability to adapt. This reflects on the adaptability requirements for the supporting software systems. Evolving a software system not only affects the source code responsible for the core application, but also the user interface. A problem with maintaining user interface (UI) logic is that it is entangled with the underlying application logic. The fact that the UI logic is scattered throughout the application logic makes adapting the UI logic and evolving the application logic cumbersome for the programmer.

In our approach we aid a programmer to cope with the complexity of UI development. To do so the UI and application code should be separated as much as possible. This idea has been applied to many software engineering areas. With respect to user interfaces, it has been applied, amongst others, by the Model-View-Controller metaphor and 3-tiered systems. Both approaches however, when put to practice, still result in entangled and scattered code and don’t help the developer to cope with several UI concerns separately.

We are currently implementing a framework to support our approach, called DEUCE. The concerns are described declaratively by means of facts and rules. A reasoning mechanism (i.e. backward and forward chainer) puts the concerns together.

The topic of this research is the documentation, verification and co-evolution of structural regularities in the source code of software. We put forward the model of Intensional Views as a generic approach to document different kinds of structural regularities as well as the interactions between such regularities. By checking conformance of these constraints with respect to the source code, we can detect evolution conflicts and provide support for co-evolving the documentation of the structural regularities and the source code of the system. To provide support for this model of intensional views, we implemented IntensiVE, a tool suite that extends the IDE of VisualWorks Smalltalk.

In order to illustrate the broad applicability of our approach, we documented a variety of different types of structural regularities and assessed to which extent they are supported by our model and tool suite. More specifically, our approach supports naming conventions, coding conventions, programming idioms, design patterns, anti-patterns and framework documentation. Our work is also applicable in the context of aspect-oriented programming. We showed that the fragile pointcut problem, an open evolution problem in the AOSD community, is caused by the tight coupling of aspect-oriented programs with the structural regularities that are prevalent in an application. To address this problem, we integrated our approach with an aspect-language and demonstrated how this approach can alleviate the fragility of pointcuts.
The recent advances in the field of Ambient Intelligence (AmI) have raised new challenges in the field of distributed garbage collection (DGC). AmI applications must deal with a more sophisticated network topology where connections are volatile and new devices can appear and disappear at any point in time. Relying on network connectivity of the devices to determine reachability of remote objects becomes impractical in this context.

This research proposes a novel treatment of distributed memory management to cope with the hardware phenomena of mobile networks. Rather than focusing on investigating a fully automatic DGC algorithm, we explore a semi-automatic garbage collection system where developers can explicitly assist the collector. Since developers have semantic knowledge of the object graph and how references are used, such meta information could be transmitted to the collector to help it to ascertain when a remote object can be reclaimed in presence of frequent disconnections. DGC is thus conceived as a high level process - rather than a merely low-level algorithm - that developers must guide when developing software for mobile networks.

Distributed garbage collection, ambient intelligence, pervasive computing annotation systems

Context-aware computing envisions scenarios in which system behaviour is parameterized by context information such as location, activity or battery level. As these context parameters change over time, context-aware systems should adapt their behaviour accordingly. Although we currently find an important amount of research on context-awareness, most of the approaches focus on the way software systems can perceive their surrounding context. We believe that this is just half-way of the development of a context-aware system. A new and totally different problem emerges when programmers start using the context information to dynamically adapt the behaviour of software systems. We claim that mainstream language constructs to implement the contextual influence on program behaviour makes the development more complex than necessary.

Pervasive computation, ambient intelligence, logic reasoning, aspect-oriented programming, collaboration-based designs
Coordination of Context-Aware Distributed Systems

Within the domain of Ambient Intelligence (AmI), I focus on the capacity of software applications to adapt to their dynamically reconfigurable environments. The main idea is to identify the properties of context-dependent adaptations and then establish some specific requirements of distribution for such adaptations, derived from concrete AmI scenarios. Context-dependent adaptations occur dynamically and within a delimited scope of action. In addition, these adaptations should be consistently combined with the default behaviour of the application, and clearly modularised to avoid the entanglement between the adaptations and the application behaviour. To cope with the effects of distribution on context-dependent adaptations, an adaptation should take into account the context of all the applications involved in an interaction, have an unambiguous scope of action even in the presence of concurrent interactions, and finally protect the privacy of the interacting applications. Currently I investigate amongst others:
- Composition of context-dependent application behaviours using prototype-based solutions
- Context-dependent adaptations of behaviour using role-based models.
- Rule-based systems for context reasoning and role selection.
- Role-based communications.
- Combining actor and role models for the development of context-dependent applications.

Context-oriented programming, ambient-oriented programming, actors, layers, roles, symbiosis

Stijn Mostinckx

High-Level Fault-Tolerance Abstractions in an Ambient-Oriented Programming Language

The far-reaching introduction of small, mobile and often dedicated computer hardware in our everyday life has a significant impact on the development of distributed systems. As these devices are becoming an integral part of our everyday environment, it becomes ever more interesting to develop software that leverages the network formed by these devices.

One of the key issues when developing such "ambient-oriented" software, is the ability for it to cope with the inevitable failures encountered as devices go in and out of communication range. Dealing with such failures requires a family of dedicated abstractions offering transactional guarantees in a fleeting environment.

The Ambient-Oriented Exception Handling Model provides a basis from which such advanced abstractions can be constructed.

Ambient-oriented programming, exception handling, asynchronous communication
Event-driven programming is an essential paradigm for implementing graphical user interfaces, business rules, web services and distributed applications because the control flow of these sorts of applications depends on the occurrence of external events. Current event-driven languages and frameworks offer no (satisfying) support for modularizing crosscutting concerns. The implementation of crosscutting concerns is scattered over different event handlers, complicating the code considerably. In addition, we observe that crosscutting concerns in event-driven applications often depend on the occurrence of multiple events in the past or future program execution.

Existing aspect-oriented languages are not suited for the event-driven paradigm, as support for expressing temporal relations between pointcuts describing events is lacking. Therefore we are researching a novel aspect language, incorporating temporal reasoning, specifically designed for event-driven programming. HALO is an aspect language offering a pointcut language based on temporal logic easing the writing of pointcuts that capture part of the execution history. In addition our approach investigates new weaving techniques based on forward chaining, pointcut analysis for discarding information about the past execution history, support for detecting and resolving aspect interaction, etc.

**Charlotte Herzeel**

*A Declarative Aspect Language, based on Temporal Logic for Event-driven Programming*

[caherzee@vub.ac.be](mailto:caherzee@vub.ac.be)

**Aspect-oriented programming, temporal-logic pointcuts, forward chaining, event-driven programming**

---

A significant body of research in ubiquitous computing deals with mobile networks, i.e. networks of mobile devices interconnected by wireless communication links. Due to the very nature of such mobile networks, addressing and communicating with remote objects is significantly more difficult than in their fixed counterparts. This research reconsiders the remote object reference concept -- one of the most fundamental programming abstractions of distributed programming languages -- in the context of mobile networks. We describe a number of requirements to which remote references in mobile networks should adhere to, show how existing remote object reference abstractions fail to do so, and postulate the need for novel referencing abstractions. Consequently, we propose ambient references: remote object references designed specifically for mobile networks. As a validation, we embed ambient references into an actor-based programming language designed specifically for mobile computing networks, named AmbientTalk. Finally, a comparative study with related work exposes the benefits and drawbacks of ambient references.

**Tom Van Cutsem**

*Ambient References - Addressing Objects in Mobile Networks*

[tvcutsem@vub.ac.be](mailto:tvcutsem@vub.ac.be)

**Pervasive and ubiquitous computing, mobile networks, remote object references, actors, distributed languages**

---

Two figures of Charlotte Herzeel and Tom Van Cutsem are present in the image, each with their respective areas of research.
Context-Oriented Programming

Context-dependent behavior is becoming increasingly important for a wide range of application domains, from pervasive computing to common business applications. Unfortunately, mainstream programming languages do not provide mechanisms that enable software entities to adapt their behavior dynamically to the current execution context. This leads developers to adopt convoluted designs to achieve the necessary runtime flexibility. Context-oriented Programming addresses this problem by treating context explicitly and providing mechanisms to dynamically adapt behavior in reaction to changes in context, even after system deployment. The research on Context-oriented Programming covers the range from the design and implementation of programming language extensions up to tool support for developing non-trivial context-aware applications.

Modularizing Advanced Transaction Management: Tackling Tangled Aspect Code

Transactions have long been a cornerstone of concurrency management in distributed systems. There are some limitations to the use of classical transactions, which are addressed by more advanced forms of transaction management. Using these proves to be difficult, however; due to their complex and cross-cutting nature. In our research we have achieved a successful modularization of this complex cross-cutting concern by designing the domain-specific aspect language KALA. In this research we have also shown that such complex aspects themselves may be composed of a collection of concerns, where some of them can be cross-cutting. In other words, the code of the aspect itself will tangle multiple concerns, a phenomenon we term tangled aspect code. Lastly we introduce the concept of syntactic obliviousness: While the programmer is aware that transactions are used in the application, no transaction management code can be seen in the base concern.
Agile Software Development (AgSD) stresses a highly iterative and incremental development cycle and rapid prototyping in order to anticipate "change" during the different phases of software engineering. There exist however a large number of unaddressed challenges in the tool support for AgSD. We focus on three related issues. First, because certain software systems, such as critical systems, can not be stopped, changes need to be performed directly on running systems. Second, testing plays a central role in AgSD. Unit tests are automated pieces of code that invoke a different method and then check assumptions on the behavior of the element currently being tested. It has to be possible to easily write the tests and quickly run them, repeatedly and automatically. Finally, automated testing gives rise to refactoring, which are continuously applied in Agile methods such as XP in order to facilitate adding new functionality or to improve the design quality after a change took place.

Current tool support for AgSD does not sufficiently support runtime change propagation, automated testing or refactorings. Representing changes as first-class objects allows a change management system for alleviating those difficulties. We extend an existing approach called Advanced Round-Trip Engineering (ARTE) -- which already provides limited support for runtime change propagation -- with such a change management system. Change-Oriented Advanced Round-Trip Engineering is an ARTE approach in which changes can be specified, applied, undone and reasoned about on each of the three ARTE levels.

Peter Ebraert

Change-oriented Advanced Round-Trip Engineering

Kris Gybels

Aspect-Oriented Programming with a SOUL

My research interests are the foundations of Logic Meta Programming (LMP) and its applications to Aspect-Oriented Software Development. Logic meta programming is the use of a programming language based on the logic paradigm (such as SOUL) for writing programs about programs, such as design recovery tools, code refactoring browsers or "pointcuts" in Aspect-Oriented Programs (AOP). While the use of a logic language for meta programs that are not necessarily written in a logic language has advantages due to the declarative nature of logic programs, it also raises a number of research questions such as how to properly represent programs as data in the logic language, and how to integrate the logic and non-logic language to create a linguistic symbiosis. Ideally, such a symbiosis is transparent, allowing interaction between the two languages while not making it obvious that a boundary between languages is crossed. I have applied this to business rule integration, where it allows the object-oriented parts of the software to be freely replaced by logic rules and vice-versa. My application of LMP to AOP revolves around the design of pointcut languages, which are used in AOP to describe which runtime events to intercept in a program. I have created an advanced pointcut language, CARMA, which, by using sophisticated features of LMP, helps decrease the coupling of these descriptions to the rest of the program. I am doing further research on the design of this language, and the question of whether techniques such as Inductive Logic Programming can be used to automatically mine programs for pointcuts.
Programming Languages for Ubiquitous Computing and Ambient Intelligence

Dr. Jessie Dedecker

Ambient-oriented programming is a programming paradigm whose properties are derived from the characteristics of hardware platforms for mobile computing. Mobile hardware devices are often provided with wireless networks facilities, allowing them to engage in collaboration with their environment. However, the autonomous nature of these devices as well as the volatile connections over their wireless infrastructure has its repercussions on the software that employs them. The most fundamental assumption of the Ambient-Oriented Programming paradigm is that languages should incorporate possible network failures at the heart of their programming model.

AmbientTalk is a distributed programming language that supports this paradigm and is designed as a language laboratory. The language offers a meta-object protocol that enables language designers to experiment with novel programming abstractions. The goal of these abstractions is to contribute to the rapid application development of Ubiquitous Computing and Ambient Intelligent applications.

jedeck@vub.ac.be

Domain and Operational Evolution in Ambient Computing

Dr. Ellen Van Paesschen

Pervasive computing embeds computer technology in smart environments and provides services and applications to any device over various kinds of networks. Ambient Intelligence extends Pervasive computing with intelligent User Interfaces that have to adapt to different contexts, in order to improve and simplify life. These intelligent User Interfaces give rise to different kinds of techniques for the personalisation or the user customisation of the mobile applications that run on mobile devices.

We observe that there is a gap between the domain level and the operational level in pervasive and ambient computing. It is our belief that this complicates evolution at both levels which we have illustrated with an example in ambient user customisation and in smart environments. We propose an intermediate reasoning level inbetween that contains a reasoning module and a synchronisation module. This enables domain evolution with the help of a new generation of end-user languages that contain static, dynamic and textual models, which is synchronised at the operational level and vice versa.

ellen.vanpaesschen@iitl.fr
Dr. Wim Vanderperren  

Combining Aspect-Oriented and Component-Based Software Engineering

The JAsCo AOP language, introduced in 2002, aims to combine aspect-oriented and component-based ideas. JAsCo introduces reusable aspect beans - aspects which are also java beans - and expressive connectors that are able to describe the aspect/component composition in detail. JAsCo also excels in providing a run-time weaver that allows dynamic aspect (un)deployment without affecting the run-time performance significantly more than its static counterparts (such as AspectJ).

A common property of component-based applications is that they are mostly distributed. This poses challenges to the aspect-oriented approach as it is originally conceived for a non-distributed setting. Therefore, current component application severs with support for AOP (such as e.g. Spring) in fact only support non-distributed AOP, which means that each host has its own aspect engine and corresponding local aspects, which are totally unaware of the distributed setting. This is in contrast to the components in such application servers, which are aware of the distributed nature and have explicit means of communicating/locating other components. To solve this, we propose the AWED language, which is an extension of AspectJ with explicit support for distribution on three levels: remote pointcuts, distributed advice and distributed aspects. The AWED technology builds on top of the JAsCo technology and as such leverages its efficient run-time weaving capability.

Bruno De Fraine  

Improving Language Facilities for the Deployment of Reusable Aspects

The aspect-oriented paradigm aims to modularize concerns that crosscut traditional abstraction boundaries. In the AOSD community, there is an increasing interest in the development of reusable implementations of typical crosscutting concerns, such as security, synchronization, profiling, etc. To employ a reusable aspect in a concrete application, deployment logic has to be written that specifies where and how to apply the new behavior, and how the interaction with the base application and the other aspects in the system is organized. Although the deployment logic might specify a crucial part of the application functionality, current AOP approaches provide only inferior means for its specification. We identify a number of issues regarding the reuse of deployments, their dynamic invocation and their integration with the rest of the system.

The Eco AOP model addresses these shortcomings by organizing deployment logic as procedures that employ reusable building blocks of aspetual behavior as first-class values. This means that pointcuts, advices and combinations strategies are passable and returnable as parameters to and from deployments, and that the deployments can be dynamically invoked with these entities as runtime values. EcoSys realizes the Eco model as a Java AOP framework, and allows the development of deployment logic as standard Java code. Contrary to other AOP frameworks, EcoSys exploits the Java 5 Generics feature to provide static type checking of deployments. As a byproduct of this research, a novel type system for the AOP pointcut/advice mechanism is being developed.
Dennis Wagelaar

Platform Ontologies for the Model Driven Architecture

The Model-Driven Architecture (MDA) allows for the deployment of software applications on a variety of different platforms. The MDA models software in a platform-independent model (PIM) that is transformed to a platform-specific model (PSM), given a platform model (PM). Currently, models are transformed directly from PIM to PSM, without using a platform model. The model transformations implicitly assume a platform model. This makes it much easier to write model transformations, since one only has to deal with the limited scope of targeting a single, assumed platform. It is unclear, however, whether a model transformation can be used for other platforms. The only safe assumption is that each targeted platform requires its own corresponding set of model transformations.

We propose a separate platform model, which can be used to automatically select and configure a number of reusable model transformations for a given platform. This platform model is expressed in OWL-DL. Platform constraints can be defined for each model transformation. This way, the model transformations can be reused over a well-defined class of platforms. Concrete platforms are modelled separately and refer to the same platform vocabulary model. An automatic DL reasoner can be used to verify whether a concrete platform satisfies the platform constraints of a model transformation. In addition, it can determine which platform constraint is most platform-specific.

dennis.wagelaar@vub.ac.be

Coen De Roover

Behavioral Program Queries using Logic Source Code Templates

The use of executable logics to query a program's implementation has gained significant momentum among researchers across software disciplines as diverse as quality assurance, program comprehension and refactoring. The efficacious identification of implementation parts causing faulty, well-known or suboptimal behavior is key to the success of logic meta programming in these disciplines. As similar behavior can be implemented by heterogeneous code, querying a behavioral rather than a syntactic program representation is a seemingly obvious strategy to improve identification efficacy. In reality, the arrival of behavioral program queries to an application programmer's toolbox is still a long way off.

First of all, it is unclear what generic behavioral representation suffices to answer a rich set of user-defined queries. As these representations inevitably approximate run-time behavior, special care must moreover be taken in answering queries. Last but not least, the representations' intricate complexity hinders straightforward query definition. The efficacious resolution and straightforward definition of behavioral queries hence requires a specifically tailored executable logic. To ensure straightforward definition, our research introduces source code templates in logic queries to capture the prototypical implementation of the behavior that needs to be identified. Logics of qualified truth are available to demarcate specific execution contexts. To ensure identification efficacy, we resolve templates against behavioral representations. The resolution strategy relies on logics of quantified truth to handle approximations and to quantify the similarity between a template and its matches.

cderoove@vub.ac.be
Inconsistency Management in Model-Driven Engineering

One of the important challenges in current-day MDE is the ability to manage model inconsistencies. When designing models in a collaborative and distributed setting, it is very likely that inconsistencies in and between the models will arise because: (i) different models may be developed in parallel by different persons; (ii) the interdependencies between models may be poorly understood; (iii) the requirements may be unclear or ambiguous at an early design stage; (iv) the models may be incomplete because some essential information may still be unknown. In a model evolution context, the ability to deal with inconsistent models becomes even more crucial, as models are continuously subject to changes.

The global objective of this research is to develop a coherent inconsistency management framework for the definition, detection and handling of inconsistencies in the context of object-oriented models with special focus on UML models. This framework enables a precise definition and detection of inconsistencies and the semi-automatic handling of inconsistencies. To obtain this global objective, we use Description Logics as a declarative formalism for inconsistency definition, detection and resolution. The reasoning capabilities of DLs can be used to reason about models.

We are currently re-implementing our framework called RACOoN which was originally implemented in the UML case-tool Poseidon, as an Eclipse plug-in.

rvdstrae@vub.ac.be

Inconsistency management, model-driven engineering, logic-based languages, software evolution

Model-Driven Engineering and Aspect-Oriented Programming for Integrating High-Level Business Rules in OO Applications

Many state-of-the-art approaches exist today that advocate making business rules explicit and separate from the object-oriented core functionality. However, we observe some problems with them: even when the decoupling of the business rules is successfully achieved, the connection code is still tangled with and scattered in the core application functionality. Secondly, executable rules are ultimately expressed in some kind of technical solution which implies the need for having programming skills, and therefore excludes the domain experts. Thirdly, a tight coupling exists between the business rules and the existing core application's implementation.

We propose using Aspect-oriented Programming for decoupling rule connections. Moreover, we identify recurrent issues that re-appear in the implementation of these connection aspects and therefore propose abstracting them as elements of aspect patterns. In order to incorporate the domain experts in the process of defining the business rules, we propose building a layer of abstraction — a domain model — on top of the existing implementation, the low-level rules and the aspects encapsulating the rule connections. The objective is to express business rules and their connections in terms of domain concepts.

We explore ideas from MDE in order to automatically and transparently translate the high-level model into implementations in OOP and AOP.

mcibran@vub.ac.be

Model-driven engineering, aspect-oriented programming, business rules, aspect patterns
In an Ambient Intelligent setting, various kinds of hardware are involved, each with its own characteristics. To allow abstraction over the different devices, a standard virtual machine approach is not feasible because it is not possible to abstract over all the devices at the same time. Instead, each device will require an individually adapted virtual machine.

This research proposes a new way to develop virtual machines for Ambient Intelligence. Instead of manually modifying the virtual machine for of a particular device, the virtual machine can be generated by composing reusable modules. The first step is to determine the set of modules that can be used to build a virtual machine. The next step is to find implementation and composition techniques that guarantee the correctness and efficiency of the generated virtual machine.

**Modularizing Language Constructs: A Reflective Approach**

Programming languages are grown with new constructs so that programmers can express the problems from their domain within the language they are using. Growing languages means to grow their implementations along with them. To support this, we preserve the decomposition of languages into language constructs in their implementations, by modularizing the definition of language constructs in separate implementation modules containing their syntactical representation and their translational semantics. The complexity of the modularization of language implementations lies in the fact that language constructs intrinsically take into account other language constructs and therefore compromise their opportunities for modularization.

We work on an open design for a new language development technique where a language implementation is decomposed into a set of interacting modular language modules called linglets. Each linglet defines in isolation the syntax and the (translational) semantics of a single language construct in terms of another (lower level) language.

The mechanisms to establish the necessary interactions among the language constructs are captured in so called strategies. The strategies are defined as extensions of a specifically tailored metaobject protocol (MOP). Since different language constructs need to communicate with each other, multiple and tailored communication mechanisms are necessary for a specific language implementation at hand. However, there is no silver bullet strategy but an open implementation brings us closer to that goal.
**Mathieu Braem**

**Aspect-Oriented Workflow Patterns for Web Service Composition**

In current composition languages for web services, there is insufficient support to explicitly separate crosscutting concerns, which leads to compositions that are hard to maintain or evolve. A similar problem in object-oriented languages is being tackled by aspect-oriented programming, and some work has been started to apply these techniques to web service composition languages as well. We identified some key problems with these approaches and formulated some improvements on the current work. We started implementing these features in Padus, an aspect-oriented language to instrument WS-BPEL, the most well-known language for web service composition.

In future work we plan to further investigate aspect-oriented programming in a workflow context and aim to define an extension to generic workflow patterns, that can in the end be mapped back to concrete languages, such as e.g. YAWL and WS-BPEL.

---

**Niels Joncheere**

**Lifting the Abstraction Level of Web Service Composition using a Visual Service Composition Environment**

Web services are an exciting new technology for integrating existing applications in rapidly evolving software systems. Using XML-based standards for interface description, messaging, and discovery, web services can be exposed to the network and interoperate with each other, no matter what the underlying implementation of these services may look like. Currently, web services are composed using dedicated web service composition languages such as WS-BPEL. Although this is a significant improvement over web service composition using general-purpose programming languages such as Java, compositions are still expressed at a relatively low level of abstraction. Our research aims to allow web service composition at a higher level of abstraction, as plug-and-play composition of individual building blocks, using a visual service composition environment (SCE), which guides the developer in creating valid compositions. The SCE allows modularizing crosscutting concerns using Padus, an aspect-oriented programming language for WS-BPEL, and provides a framework for developing concern-specific languages on top of Padus. The SCE also performs quality-of-service verification on the compositions that are being created.
Web service technology is an open standards-based mechanism for communication over a network regardless of the hardware or platform used on either side of the wire. As Web services enable computer-to-computer communications in a heterogeneous environment, they are ideally suited for the Internet. We have observed that the implementation and deployment of a service-oriented architecture (SOA) where several third-party Web services need to integrated, poses a variety of challenges. More specifically, current approaches fall short in providing the needed runtime flexibility to deal with various events in a dynamic services environment.

In our approach, we deploy a mediation framework that takes care of all service related concerns in a SOA. The framework, called Web Services Management Layer (WSML) introduces a flexible service redirection mechanism that allows for the run-time integration of services and compositions while taking into account non-functional selection policies and enforcing a range of client-side management concerns. As a lot of code dealing with the various identified service concerns results crosscutting in the SOA at those places where a service functionality is required, we opt to employ dynamic Aspect-Oriented Programming (AOP) to achieve a better separation of concerns. The WSML offers support during the development, deployment and runtime cycle of a SOA. To avoid that WSML administrators require aspect-oriented knowledge, the use of aspects is hidden by using aspect libraries and by doing automatic generation of aspect code.
<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>for Dynamic Languages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Concept-Centric Environment for Software Evolution in an Agile</td>
<td>Dirk Deridder</td>
<td>2006</td>
</tr>
<tr>
<td>Context</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ambient-Oriented Programming</td>
<td>Jessie Dedecker</td>
<td>2006</td>
</tr>
<tr>
<td>Modularizing Advanced Transaction Management</td>
<td>Johan Fabry</td>
<td>2005</td>
</tr>
<tr>
<td>Inconsistency Management in Model-Driven Engineering: An Approach</td>
<td>Ragnhild Van Der Straeten</td>
<td>2005</td>
</tr>
<tr>
<td>using Description Logics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrative Composition of Program Generators</td>
<td>Johan Brichau</td>
<td>2005</td>
</tr>
<tr>
<td>Combining Aspect-Oriented and Component-Based Software Engineering</td>
<td>Wim Vanderperren</td>
<td>2004</td>
</tr>
<tr>
<td>Hybrid Aspects for Integrating Rule-Based Knowledge and Object-</td>
<td>Maja D’Hondt</td>
<td>2004</td>
</tr>
<tr>
<td>Oriented Functionality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Considered Harmful: A Language Design Approach to Mobility and</td>
<td>Wolfgang De Meuter</td>
<td>2004</td>
</tr>
<tr>
<td>Distribution for Open Networks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Progressive Mobility</td>
<td>Luk Stoops</td>
<td>2004</td>
</tr>
<tr>
<td>Creation of an Intelligent Concurrency Adaptor in order to mediate</td>
<td>Werner Van Belle</td>
<td>2003</td>
</tr>
<tr>
<td>the Differences between Conflicting Concurrency Interfaces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Automated Support For Framework-Based Software Evolution</td>
<td>Tom Tourwe</td>
<td>2002</td>
</tr>
<tr>
<td>Component Composition using Composition Patterns and Scenarios</td>
<td>Bart Wydaeghe</td>
<td>2001</td>
</tr>
</tbody>
</table>
PhD Dissertations

UML – SDL Round-trip Engineering through Incremental Translation of Changes
Kurt Verschaeve, 2001

A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation
Roel Wuyts, 2001

Automating Architectural Conformance Checking by means of Logic Meta Programming
Kim Mens, 2000

Multi-Paradigm Design
James O. Coplien, 2000

A Formal Foundation for Object-Oriented Software Evolution
Tom Mens, 1999

An Approach to Incremental Fuzzy Modelling of Dependencies in Complex Physical Systems
Kamreddine Ouliddren, 1998

A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems
Koen De Hondt, 1998

Type-Oriented Logic Meta Programming
Kris De Volder, 1998

Documenting Reuse and Evolution with Reuse Contracts
Carine Lucas, 1997

ZYPHER - Tailorability as a Link from Object-Oriented Software Engineering to Open Hypermedia
Serge Demeyer, 1996

Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks
Patrick Steyaert, 1994

The synthesis of “safe” fuzzy controllers based on reinforcement learning
Ann Nowé, 1994

From Specification to Implementation
Ludo Cuypers, 1993