Towards Context-Sensitive Intelligence

Holger Mügge[1], Tobias Rho[1], Marcel Winandy[1], Markus Won[1], Armin B. Cremers[1], Pascal Costanza[2], and Roman Englert[3]

[1] Institute of Computer Science III, University of Bonn, Römerstr. 164, 53117 Bonn, Germany
{muegge, rho, winandy, won, abc}@iai.uni-bonn.de

[2] Programming Technology Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
pc@p-cos.net

[3] Deutsche Telekom Laboratories, Ernst-Reuter-Platz 7, 10587 Berlin, Germany
Roman.Englert@telekom.de

Abstract. Even modern component architectures do not provide for easily manageable context-sensitive adaptability, a key requirement for ambient intelligence. The reason is that components are too large – providing black boxes with adaptation points only at their boundaries – and to small – lacking good means for expressing concerns beyond the scope of single components – at the same time. We present a framework that makes components more fine-grained so that adaptation points inside of them become accessible, and more coarse-grained so that changes of single components result in the necessary update of structurally constrained dependants. This will lead to higher quality applications that fit better into personalized and context-aware usage scenarios.

1 Introduction

Most of the software sold nowadays are off-the-shelf products designed to meet the requirements of very different types of users. One way to meet these requirements is to design software that is flexible in such a way that it can be used in very different contexts. Thus, look and feel, functionality, and behavior have to be tailorable or even adaptive according to the task that needs to be fulfilled. Especially out of an organizational context most users have to tailor their software on their own. Taken into account that experiences in the use of computer systems in general increase exceedingly, tailorable and end user development applications become interesting topics. Component architectures were basically developed with the idea of higher reusability of parts of software. Furthermore, it is shown that they also build a basis for highly flexible software [1]. In this case the same operations that are used to compose software out of single components now can be applied to existing (component-based) software during runtime. Therefore, the basis for a tailoring language consists basically of three kinds of operations: choosing components, parameterizing them, binding them together. In this way very simple operations that can be easily understood by end users enhance the possibilities of tailoring software in a powerful way (cf. [2]).

2 Intelligent Dealing with Complexity

Still, there are several open questions according to how tailoring can be eased for end users. For instance, there is a need for a graphical front end that allows visual tailoring techniques. Here one problem is how invisible components can be presented to the users. In different studies it was shown (cf. [1]) that users are able to tailor their GUIs very easily. Nevertheless, the problem of finding an appropriate visual tailoring environment for both - visible and invisible components - is still unsolved.

A second problem is that tailoring becomes harder when more flexibility is needed. Flexibility in component architectures designed for tailorable applications is reached by a higher degree of decomposition [1]. That means, the more components are needed to design software, the more flexible it can be tailored as there are many fine-grained components that can be parameterized or exchanged.

Our goal is to design a stable basis for highly flexible software systems. Component architectures are appropriate in this case and thus concentrating on the second problem.

There are several approaches which may ease the use of software in different contexts. In our case we believe in a combination of tailorable software (user is in an active role) and adaptive techniques (software does adaptions by itself). This might be helpful to cope with the complexity problem. Combining both techniques means that tailoring activities are followed by automatic adaptions of the system which checks for dependencies within the composition and adjusts it.

Another point is the inspection of contexts: How do contexts look like and how can they influence the software system? The abstraction of different use contexts and their explicit description can reduce complexity of the components as context descriptions influence more the whole composition. If the context changes, users have only to switch the current context description which leads to changed functionality of the whole composition.

Furthermore, one source of complexity is that many applications run distributed and networked. In such systems (client-server, peer-2-peer) tailoring becomes even harder as adaptations on one client or one server might have dependencies on another part of the application which runs on a different machine. In such cases server components have to behave according to different clients. In section 3 we describe three basic techniques which can overcome these problems. After that we show how they can be integrated within one component framework in section 4.

3 Contributing Parts

We build our framework for CSI on top of three pillars: Generalized aspect weaving (3.1), adaptation of thread-local features (3.2), and structure elicitation and constraint checking (3.3).

3.1 LogicAJ

Modularization on component level fails on *crosscutting concerns*[3]. These can not be located in one place and are therefore scattered over several components. Examples are persistence, distribution[4], security[1][5], synchronization and parallelization. Aspect languages modularize crosscutting concerns in a new construct: aspects. Aspects keep the code belonging to the concern in one place and describe where the code should be woven into the base program.

The following properties are necessary for an aspect language in an adaptive environment:

Expressiveness: In an evolving environment aspects must deal with an unanticipated structure of components and types. Aspects must therefore be highly generic to be independent from the lexical structure of a base program.

Static Type Safety: Expressiveness should not come with a loss of static type safety. Runtime checks and reflective techniques should not be used to avoid runtime errors in aspect execution and weaving[2].

Dynamic: Aspects need to be applicable and removable at runtime to react on changes of the application context.

Current aspect languages refer to fixed names for concrete entities of the base program, where reusable implementations would require role names that can be bound to concrete entities when the pattern (resp. aspect) is instantiated. Therefore, these implementations must be modified for every program and program modifications.

A generic aspect language allows aspects to use logic variables that can range over syntactic entities of the host language. In a Java-based generic aspect language, for instance, logic variables could match anything from packages and types down to individual statements, modifiers and throws-declarations. In particular, it is possible to create new code based on previous matches. In this respect, logic variables are more expressive than "*" pattern matching (e.g in AspectJ), where two occurrences of "*" do not represent the same value.

The modularization of crosscutting concerns by dynamic aspects enables the application to be configurable at runtime. Dependent on the context different aspects adapt the application.

[1] authentication, secure socket code, ...
[2] the application of the aspect to the base program.

For example consider a client application connected to a server with sensitive data. Depending on the current network connection different aspects are woven: If the connection is unsecured[3] the simple socket code is replaced with a ssl implementation. After detecting poor network performance a caching aspect is applied. Now the user maximizes the application window. An aspect adapts the content and shows based on the profile of the user more detailed information on the current task.

In a distributed environment these changes affect components which may also be used by other components in a different context. The next section (3.2) shows how we can deal with different component adaptations at the same time.

3.2 Dynamic Scoping

A definition is said to be dynamically scoped if at any point in time during the execution of a program, its binding is looked up in the current call stack as opposed to the lexically apparent binding in the source code of that program. The latter case is referred to as lexical scoping. An important property of dynamic scoping is that it fits naturally with multi-threaded programs when the new binding to a dynamically scoped variable is restricted to the current thread. Almost all programming languages in wide use employ lexical scoping but do not offer dynamic scoping. Notable exceptions are Common Lisp, various Scheme implementations, and recent attempts at introducing dynamic scoping into C++ [6] and Haskell [7].

We have achieved a similar level of usefulness when adding dynamic scoping for function definitions, and we have described two different working implementations of that idea in [8, 9]. A similar extension for Java looks as follows. Assume we have a method in a mobile application that performs an operation which may cause a large cost on the user's side. For example, it contacts a payed service on the network. The user may be interested in that method behaving differently depending on various contexts. For example, it should pop up a dialog that asks for authentication first for security reasons, it should simulate some useful response from the service in order to explore the possibilities, or it should just prevent the method from executing at all when lending the mobile device to some other user. An activation of such context-specific behavior looks as follows:

```
with {
  contactExpensiveService () {
    if askUser("Are you sure?") proceed();
    else throw new ServiceException();
  }
}{
  runApplication();
}
```

[3] For example, the connection uses an untrusted network and no VPN connection is active.

An important ingredient to make such context-specific behavior work is the `proceed` command that executes the original definition of the redefined method. This is reminiscent of `proceed` in AspectJ [3] and `call-next-method` in CLOS. Without such a `proceed` command, a dynamically scoped redefinition would only be able to completely replace an existing method (and this is an important difference to dynamically scoped variables).

According to the code above, the new definition for `contactExpensive Service`first asks the user for an acknowledgement and then either proceeds with execution of the original method or rejects its execution. Without a notion of dynamically scoped methods, such a behavior modification would only be possible by inserting appropriate `if` statements into the base code, leading to error-prone and hard-to-maintain code. With dynamically scoped methods, all the different contexts are cleanly separated within their threads and can be modified independently from each other. So in the case of method definitions, dynamic scoping again helps to avoid cluttering code with context-specific behavior.

3.3 Structural Constraints

Complex software usually comprises many variation points each with a number of different variants (as defined by van Gurp et al. in [10]). As a result increasing adaptability leads to a combinatorial explosion of potential adaptations. Separation of concerns as shown in sections 3.1 and 3.2 prevents the code from being polluted by scattered and intransparent conditional statements. But nevertheless the complexity shows up when the variation points are designed or the software is going to be configured, i.e. when one particular adaptation scenario has to be chosen. We need to know which variants can or should be combined and which must not or what consequences a certain adaptation implies.

This knowledge is not contained in the software per se. It is meta-information derived from application semantics or technical context for example and usually only given implicitly as Dolstra et al. point out in [11]. To tackle these issues, we integrate the framework PatchWork which allows for modelling complex structural conditions as explicit meta-data. It enables defining structure schemata relations between role sets with constraints imposed on them. Instances of a structure schema can be checked against these relational constraints and the software composition can be guided by the relational structure.

We demonstrate the usage of such structural meta-data with the following scenario: assume our software system comprises three functionalities a, b and c and offers the user two different and complete ways to access them, *menu-driven* and via *key shortcuts*. Both the functionality set and the set of access ways are configurable, hence they represent variation points [10].

Now, we want to enhance our software with a third way of user access via *speech recognition*. Figure 1 illustrates the performed adaptations. The table shows an instance of an underlying structure schema defining three role sets

Functionality (F)	User Access (U)		**New way of access**
	menu-driven	key shortcuts	**speech recognition**
a	$i_{m,a}$	$i_{k,a}$	$\mathbf{i_{s,a}}$
b	$i_{m,b}$	$i_{k,b}$	
c	$i_{m,c}$	$i_{k,c}$	$\mathbf{i_{s,c}}$

Fig. 1. Structural meta-data in the background supporting software adaptation

(functionalities F, user access ways U and implementations I), a ternary relation R between them, and a constraint on R which guarantees accessibility of each function by each user access way. The initial situation is depicted with solid lines in the table. Initially holds $F = \{a, b, c\}$, $U = \{\textit{menu-driven}, \textit{key-shortcut}\}$, and $I = \{i_{m,a}, i_{m,b}, i_{m,c}, i_{k,a}, \ldots\}$.

First, we add speech recognition as a new user access way. It is represented in our meta-data as a new player for the *user-access-ways role*, i.e. *speech-recognition* $\in U$. The underlying structure schema declares R to be total in $F \times U$, so we are forced to add valid tuples for each functionality to R, as shown in figure 1 with dotted lines.

In a *second* step we realize that functionality b is too complex to be accessed by speech recognition (i.e. there will be no element $i_{s,b}$ shown as the empty grey cell in figure 1). Hence, we must loosen our initial totality condition on R to allow that speech recognition does not offer access to all functionalities. We adjust the structure schema so that the constraint on R only demands totality in F so that it still guarantees at least one way of access for each functionality. Even though this may look like we simply give up a part of our initial requirements, we are forced to do that explicitly and consciously.

Finally as a *third* step we consider what happens when the usage context allows only to use speech recognition, i. e. $U = \{\text{speech-recognition}\}$. Now, totality of R in F is violated because the required tuple for b is missing. This could for example lead to automatically reducing the functionality set to $F = \{a, c\}$ so that the structural constraints hold again.

In reality the situation quickly gets more complex since one has to take into account contextual aspects like: device capabilities, access rights, location etc., therefore explicit modelling structural constraints becomes even more useful.

4 Summary

Our three approaches provide the following additional means to influence the behavior of component-based applications (see Figure 2): (1) LogicAJ helps im-

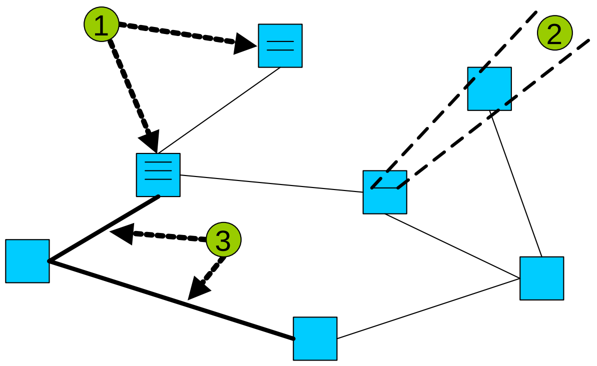

Fig. 2. Our approaches for an advanced component framework: (1) Generic aspect-oriented programming allows dealing with advanced crosscutting concerns; (2) dynamic scoping allows influencing the program's behavior from certain contexts without interfering with others; (3) PatchWork allows expressing and controlling structural constraints

plementing aspects that implement crosscutting concerns and are highly generic to be applicable in different contexts. (2) Dynamically scoped methods provide a mechanism for behavioral changes of an application that can be confined to thread boundaries without affecting other threads, leading to a natural mapping of contexts to threads. (3) Finally, elicitation of structural constraints and automatic checking of such constraints ensure that local changes to single components either do not violate coarse-grained, non-localizable dependencies, or else even trigger the subsequent automatic correction of dependents to adapt to the new environment.

These different approaches already substantially improve context-sensitive adaptability. However, a combination of these approaches has interesting synergistic effects: Dynamically scoped methods can be extended towards dynamically scoped activation of generic aspects while structural constraints are automatically maintained beyond traditional component-based means of adaptation. We have carried out first experiments to see whether our approaches indeed complement each other in this way, and the results thereof are very promising.

Security of adaptation is another important issue because new functionality may come from an untrustworthy source. Existing protection mechanisms must not be corrupted or by-passed and missing mechanisms should be added automatically.

Future work includes building a stable and secure software architecture that incorporates the ideas that we have sketched in this paper on the technology side, and carrying out user studies to understand to what extent end users are capable of dealing with our new abstractions.

References

1. Morch, A.I., Stevens, G., Won, M., Klann, M., Dittrich, Y., Wulf, V.: Component-based technologies for end-user development. Communications of the ACM **9** (2004) 59–66
2. Won, M., Stiemerling, O., Wulf, V.: Component-based approaches to tailorable systems. In Lieberman, H., P.F., Wulf, V., eds.: End User Development. Kluwer Academic (2005)
3. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary. Volume 2072 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (2001) 327–353
4. Silaghi, R., Strohmeier, A.: Better generative programming with generic aspects. (2003) 2nd International Workshop on Generative Techniques in the Context of MDA, OOPSLA 2003, Available as Technical Report, N IC/2003/80, Swiss Federal Institute of Technology in Lausanne, Switzerland, December 2003.
5. Viega, J., Bloch, J., Chandra, P.: Applying aspect-oriented programming to security. Cutter IT Journal **14** (2001) 31–39
6. Hanson, D.R., Proebsting, T.A.: Dynamic variables. In: Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, Snowbird, Utah (2001) 264–273
7. Lewis, J.R., Shields, M., Launchbury, J., Meijer, E.: Implicit parameters: Dynamic scoping with static types. In: Symposium on Principles of Programming Languages, ACM Press (2000) 108–118
8. Costanza, P.: Dynamically scoped functions as the essence of aop. In: ECOOP 2003 Workshop on Object-Oriented Language Engineering for the Post-Java Era, Darmstadt, Germany, July 22, 2003, ACM Press (2003)
9. Costanza, P.: A short overview of AspectL. In: European Interactive Workshop on Aspects in Software (EIWAS'04), Berlin, Germany. (2004)
10. van Gurp, J., Bosch, J., Svahnberg, M.: The notion of variability in software product lines. In: Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). (2001) 45–55
11. Dolstra, E., Florijn, G., Visser, E.: Timeline variability: The variability of binding time of variation points. In van Gurp, J., Bosch, J., eds.: Workshop on Software Variability Modeling (SVM'03). Number 2003-7-01 in IWI preprints, Groningen, The Netherlands, Reseach Institute of Computer Science and Mathematics, University of Groningen (2003)