# Taming Obliviousness in Aspects using Data-flow Analysis and Design by Contract

Tim Molderez*, Dirk Janssens

University of Antwerp, Belgium
tim.molderez@ua.ac.be, dirk.janssens@ua.ac.be

## 1 Introduction

Aspect-oriented programming (AOP) [3] allows for the modularization of crosscutting concerns, which are those concerns that are typically scattered across several places in source code and are inherently tangled together with other concerns. An AOP language accomplishes this goal by encapsulating crosscutting concerns in so-called aspects. The key feature of an aspect is that it can implicitly decide for itself that a certain body of code should be executed at certain points during a program's execution. It is well-known in the AOP community that this implicitness both forms the main strength and the main weakness of AOP. On the one hand, a crosscutting concern can be described in a very compact manner: Instead of scattering similar-looking code across several places, an aspect can contain that code inside an advice body, combined with a pointcut expression that specifies when the advice should be executed. On the other hand, everyone but the aspect itself is *oblivious* towards the aspect's existence. This property of AOP makes an aspect-oriented application more prone to unexpected, and potentially undesired, behaviour. More specifically, the application will inevitably continue to evolve and those who are unaware of aspects can then inadvertently turn a cooperating aspect into one that causes undesired behaviour. Even disregarding evolution, the developers who write aspects need to tread carefully: Because an advice can be applied at a multitude of points during program execution, it is not always clear what effects an advice will have.

## 2 Approach

In this extended abstract, we will briefly present our current progress in solving the problems associated with obliviousness. The end goal is that we develop an approach that allows the developer to identify undesired behaviour caused by aspects in a quick, precise and easy-to-understand manner. To achieve this target, we have chosen to tackle obliviousness from two different perspectives, which already are useful individually, but will later join to meet the end goal. These two perspectives, as indicated by the title, are data-flow analysis and design by contract.

**Data-flow analysis -** First, using data-flow analysis, our aim is to make the interactions caused by aspects visible, both to the developers writing aspects, as well as to those developers being affected by aspects. Moreover, it is important that the analysis fits in seamlessly with the workflow of developers. This primarily means that we should continuously update which interactions are found as the system evolves, such that the developer can investigate these interactions at any time. It should also be easy for the developer to identify how an interaction was found, and whether it is relevant to his/her work. As such, our analysis is a static and incremental data-flow analysis. Our focus is to find all data

* Funded by a doctoral scholarship of the Research Foundation, Flanders (FWO)

dependencies caused within all potential control flows of each advice, as such data dependencies form the most interesting and non-trivial kind of interactions to study. The basic analysis algorithm, which is initially applied to the entire system, consists of two phases: For each advice, a depth-first traversal is performed to find all statements where a variable is modified, which might have a non-local effect. Second, for each variable modification that was found, we wish to find out who first uses that variable after the advice is executed. In other words, we want to find out who is affected by the modification made by that advice. We can then reuse the same algorithm to keep the results updated incrementally. That is, whenever the developer changes the source code, we can determine which parts of the code need to be reanalysed, if any. A proof-of-concept of this algorithm is currently being built on top of the AspectBench Compiler (abc) [1]. Validating this work will primarily focus on measuring time/memory performance, as well as the number of false positives (i.e. precision), using existing realistic-sized AOP applications. This validation is done both for the algorithm on the system in its entirety, as well as on large sets of random, though realistic, changes on the system to validate incremental performance.

**Design by contract** - The second perspective in tackling obliviousness is found in the area of design by contract. The aim here is to constrain aspects such that aspects are still allowed to interact, but to ensure that harmful interactions cannot occur. To accomplish this goal we have studied how contracts of advice interact with other contracts in the system [4]. We showed that, if an advice adheres to the so-called "advice substitution principle", it will not violate the contracts of other components. As a consequence, the interactions caused by advice will not cause any undesired behaviour elsewhere. This advice substitution principle is quite similar to Liskov substitution: Where overriding methods should take into account the contracts of the overridden method, an advice should take into account the contracts of those points during the execution that it intercepts. A tool, built using aspects, is also provided that is able to dynamically enforce the advice substitution principle on AspectJ [2] applications. If a contract is broken, this tool can as well correctly determine who is to blame, based on Liskov and advice substitution.

**Combining data-flow analysis with contracts** - To meet the end goal of detecting undesired interactions caused by aspects, the work in data-flow analysis and design by contract is to be combined: Our data-flow analysis is able to detect the interactions caused by aspects. This information by itself is however not sufficient to tell desired interactions from undesired ones. On the other hand, the advice substitution principle can constrain advice such that it can only perform harmless interactions. If an advice would however violate the principle, we would like to identify which interaction(s) exactly is/are responsible for this violation. These observations already hint at how the work in both areas can be combined, which boils down to answering the following question: Given an interaction caused by an aspect, will this interaction lead to a contract being broken? That is, assuming this contract was satisfied in the absence of that aspect. Validating the answer to this question, which may be a static or a dynamic technique (or a combination of both), leads back to evaluating how quick, precise and easy-to-understand the approach is. This can be done by precisely specifying which cases the approach is or is not able to handle, as well as by measuring performance and precision metrics on a realistic-sized aspect-oriented application that is augmented with contracts.

## References


1. Pavel Avgustinov, Aske Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc : An extensible AspectJ compiler. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer Berlin / Heidelberg, 2006.
2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings European Conference on Object-Oriented Programming 2001*, volume 2072 of *LNCS*, page 327–353. Springer-Verlag, 2001.
3. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, page 220–242. Springer, 1997.
4. Tim Molderez and Dirk Janssens. Design by contract for aspects, by aspects. In *11th Workshop on Foundations of Aspect-Oriented Languages (FOAL), co-located with AOSD 2012*, Potsdam, Germany, 2012.