Feature Descriptions for Context-oriented Programming

Pascal Costanza*, Theo D'Hondt*

*Programming Technology Lab
Vrije Universiteit Brussel, B-1050 Brussels, Belgium
(pascal.costanza|tjdhondt)@vub.ac.be

Abstract

In Context-oriented Programming (COP), programs can be partitioned into behavioral variations expressed as sets of partial program definitions. Such layers can be activated and deactivated at runtime, depending on the execution context. In previous work, we identified the need for application-specific dependencies between layers, and suggested an efficient reflective interface for controlling such dependencies. However, that solution requires knowledge about complex low-level details of a particular COP implementation, which can be hard to master. In this paper, we show how feature diagrams can be naturally mapped onto COP by integrating the Feature Description Language. Since this mapping ensures that feature dependencies are automatically enforced, programmers can focus on declarative descriptions of layer dependencies without the need to resort to low-level details.

I. Introduction

In Context-oriented Programming, programs can be partitioned into behavioral variations, which consist of partial class and method definitions that can be freely selected and combined at runtime. This enables programs to change their behavior according to their context of use. In [5], we have introduced this idea and have presented the programming language ContextL, which is our first language extension that explicitly realizes this vision.

In [7], we have introduced a reflective facility for coordinating and controlling the activation and deactivation of layers. It enables expressing complex application-defined dependencies between layers where the activation or deactivation of one layer requires the activation or deactivation of another one. Despite of the flexibility of that reflective facility, we have been successful to retain the efficiency of a previously described implementation technique for ContextL [6]. However, the implementation of reflective extensions to ContextL is complex: A programmer has to define new layer classes, decide which layers are instances of which layer classes, define methods on the correct hooks of the reflective API, and understand the interactions between several such methods.

In this paper, we present a declarative extension for describing feature expressions and constraints on layers, based on the Feature Description Language (FDL) [9]. Our adaptation of FDL greatly simplifies expressing dependencies between layers in ContextL: Layer dependencies can be specified as straightforward feature expressions and constraints alongside the layer definitions themselves.

II. Feature Dependencies

We illustrate possible dependencies between layers in Context-oriented Programming with different tariff plans users of mobile devices, such as PDAs and smartphones, can choose from. For example, regular phone calls, SMS, internet connection, and so on, are billed for using different tariff plans based on time, volume, flat rates, and so on. Accordingly, there should be means to control the activation and deactivation of corresponding features: A user may be interested in blocking the activation of certain premium features, or switching from one tariff plan to another without the risk of two tariff plans being active at the same time. On the other hand, a service operator may want to ensure that the activation of certain features makes the selection and activation of a tariff plan mandatory.

A. ContextL in a Nutshell

Layers are the essential extension provided by ContextL and are a good match to represent such features whose

activation state can be determined only at runtime.[1] Layers can be defined with `deflayer`, for example like this:

```
(deflayer phone-call-layer)
```

Layers have a name, and partial class and method definitions can be added to them. There exists a predefined root layer that all definitions are automatically placed in when they do not explicitly name a different layer.

For example, consider the following interface in ContextL for making phone calls:

```
(define-layered-function start-call (nr))
(define-layered-function end-call ())
```

This defines two generic functions, one taking a phone number as a parameter and the other one not taking any parameters. A default implementation for these as yet abstract functions can be placed in the root layer:

```
(define-layered-method start-call (nr)
  (error "Phone calls inactive."))

(define-layered-method end-call ()
  (error "Phone calls inactive."))
```

Only if the `phone-call-layer` is active, a user can actually make phone calls:

```
(define-layered-method start-call
  :in-layer phone-call-layer (nr)
  ... actual implementation ...)

(define-layered-method end-call
  :in-layer phone-call-layer ()
  ... actual implementation ...)
```

Layers can be activated in the dynamic scope of a program:

```
(with-active-layers (phone-call-layer)
  ... contained code ...)
```

Dynamically scoped layer activation has the effect that the layer is only active during execution of the *contained code*, including all the code that the *contained code* calls directly or indirectly. Layer activation can be nested, which means that a layer can be activated when it is already active. However, this effectively means that a layer is always active only once at a particular point in time, so nested layer activations are just ignored. This also means that on return from a dynamically scoped layer activation, a layer's activity state depends on whether it was already active before or not. In other words, dynamically scoped layer activation obeys a stack-like discipline. Likewise, layers can be deactivated with a similar `with-inactive-layers` that ensures that a layer is not active during the execution of some contained code.

[1]ContextL is an extension to the Common Lisp Object System (CLOS, [3]), which in turn is based on generic functions instead of the more wide-spread class-based object model. However, the context-oriented features of ContextL are conceptually independent of CLOS, and we have already mapped them to several other languages, including Java and Smalltalk [11]. ContextL can be downloaded at http://common-lisp.net/project/closer in the ContextL section.

Multiple layers can contribute to the same layered functions. For example, a metering layer can determine the cost of a phone call:

```
(deflayer phone-tariff-a)

(define-layered-method start-call
  :in-layer phone-tariff-a :after (nr)
  ... record start time ...)

(define-layered-method end-call
  :in-layer phone-tariff-a :after ()
  ... determine cost a ...)

(deflayer phone-tariff-b)

(define-layered-method start-call
  :in-layer phone-tariff-b :after (nr)
  ... record start time ...)

(define-layered-method end-call
  :in-layer phone-tariff-b :after ()
  ... determine cost b ...)
```

The user can now be asked to select one of the tariffs:

```
(let ((tariff (ask-user "Select tariff...")))
 (if ((equal tariff 'phone-tariff-a)
       (with-active-layers (phone-tariff-a)
         ... ))
      ((equal tariff 'phone-tariff-b)
       (with-active-layers (phone-tariff-b)
         ... ))
      ... ))
```

We can now recast the desire to control feature activation in more technical terms: The unconditional layer activation and deactivation constructs could be guarded by `if` statements that check whether a layer may be activated or deactivated, and also whether other layers should be activated and/or deactivated as a consequence as well. While technically possible, this has negative implications for maintainability, understandability, and so on, by scattering such checks throughout a program. The goal of our work is to avoid such scattering as far as possible.

B. Reflective Layer Activation

In principle, ContextL already provides control of layer activation and deactivation without polluting a program with `if` statements: Whenever a layer is activated via `with-active-layers` (or other layer activation constructs), ContextL calls the generic function `adjoin-layer-using-class` to determine the effective ordered set of layers to be active. Likewise, `with-inactive-layers` calls the generic function `remove-layer-using-class`, also to determine an effective ordered set of layers to be active. The default methods on `adjoin-layer-using-class` and `remove-layer-using-class` implement the semantics of layer activation and deactivation as described in Sect. II-A. Since `adjoin-layer-using-class` and

`remove-layer-using-class` are generic, they provide a *reflective interface* to ContextL which allows user programs to define methods for their own *layer classes* that control under what circumstances specific layers may be activated or deactivated, and whether they imply further activation or deactivation of other layers [7].

There are advantages and disadvantages in the design of ContextL's reflective API. On the one hand, it balances extensibility and performance: With an appropriate caching scheme, reflective layer activation is as efficient as without such a reflective facility [7]. On the other hand, reflective extensions are very complex: A programmer has to define layer classes, decide which layers are instances of which layer classes, define correct methods on `adjoin-layer-using-class` and `remove-layer-using-class`, and understand the complex interactions between different such methods.

In other words, while reflective layer activation is an effective hook for enforcing dependencies between layers, the actual implementation effort requires knowledge about too many low-level details. What is needed instead is a declarative interface at a higher level of abstraction to express rules for layer activation/deactivation and layer dependencies. At the same time, the advantages of ContextL's reflective API, especially in terms of its efficiency, should be retained as much as possible.

III. Feature Descriptions for ContextL

Context-oriented Programming has a lot in common with the mixin layers approach of feature-oriented programming [16]: Both approaches express program variations as layers, but in the former they are activated dynamically, and in the latter they are statically composed. An established approach for expressing feature dependencies in feature-oriented programming is the use of feature diagrams [8], [12], a graphical notation for depicting feature dependencies. The Feature Description Language (FDL) is a textual notation for such feature diagrams [9]. Since a textual notation is more straightforward to carry over to a Lisp-based language such as Common Lisp, and since the formalization of FDL in [9] yields precise semantics for feature diagrams, we have based the subsequently described extension to ContextL on FDL.

A. Atomic and Composite Features

An FDL description consists of atomic and composite features. Composite features are defined using a feature expression language which allows specifying mandatory, optional, alternative and non-exclusive selections of features expressed as `all-of`, `optional`, `one-of` and `more-of` respectively.[2] Each atomic feature can specify constraints by requiring or excluding other atomic features. Moreover, users can define further constraints by unconditionally including and excluding specific atomic layers. These constructs are mapped to ContextL as follows.

a) Atomic features: Atomic features are mapped to atomic layers, which behave mostly like plain layers in ContextL – they can be activated and deactivated with dynamic scope. In addition to that, they can require or exclude the presence of other atomic layers. For example, we can define the atomic layers `phone-tariff-a` and `phone-tariff-b` as follows:

```
(define-atomic-layer phone-tariff-a
  (:requires phone-call-layer)
  (:excludes phone-tariff-b))

(define-atomic-layer phone-tariff-b
  (:requires phone-call-layer)
  (:excludes phone-tariff-a))
```

The next fragment shows how we can express mutual exclusions, but they are internally mapped to exclusion specifications on atomic layers, as in the example above:

```
(define-mutual-exclusion
 phone-tariff-a
 phone-tariff-b
 phone-tariff-c)
```

b) Composite features: Composite features are mapped to composite layers, which are defined together with their feature expressions.[3] The following example composite layer definitions correspond to the feature diagrams in Fig. 1:

```
(define-composite-layer phone-tariff
  (one-of phone-tariff-a
          phone-tariff-b
          phone-tariff-c))

(define-composite-layer flat-rate-option
  (more-of flat-rate-1 flat-rate-2))

(define-composite-layer internet-tariff
  (all-of (one-of phone-tariff-b
                  phone-tariff-c)
          (optional flat-rate-option)))
```

While atomic layers can refer only to other atomic layers in their constraints, composite layers can refer to both atomic and other composite layers in their feature expressions.

In [9], the semantics of feature expressions are defined using a feature diagram algebra that can be used to automatically transform a feature expression to its disjunctive normal form (DNF). For example, the algorithm described in [9] yields the following DNFs for `phone-tariff` and `internet-tariff`:

[2] In [9], these are expressed as `all`, `?`, `one-of` and `more-of`. Non-exclusive selections are sometime also called "or-features" [8].

[3] In this paper, we use a notation for feature expressions based on Lisp-style s-expressions. In [9], a more conventional syntax is used.

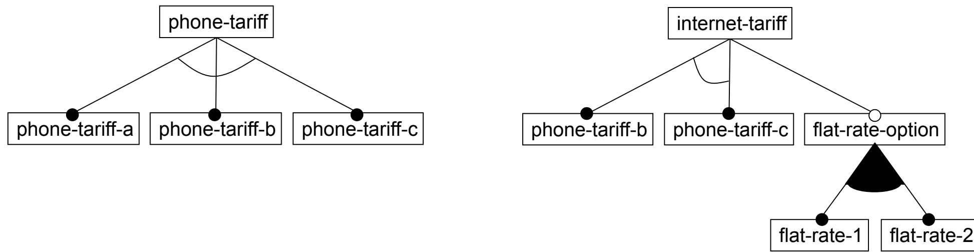

Fig. 1. Example feature diagram

```
;; phone-tariff
(one-of (all-of phone-tariff-a)
        (all-of phone-tariff-b)
        (all-of phone-tariff-c))

;; internet-tariff
(one-of (all-of phone-tariff-b)
        (all-of phone-tariff-b flat-rate-1)
        (all-of phone-tariff-b flat-rate-2)
        (all-of phone-tariff-b
                flat-rate-1 flat-rate-2)
        (all-of phone-tariff-c)
        (all-of phone-tariff-c flat-rate-1)
        (all-of phone-tariff-c flat-rate-2)
        (all-of phone-tariff-c
                flat-rate-1 flat-rate-2))
```

It is important that such DNFs can indeed be automatically and unambiguously determined, because they yield all valid feature alternatives that satisfy the respective feature expression. This makes FDL's feature diagram algebra suitable for automatic validity checks.

c) Constraints: There is no direct mapping of user constraints (unconditional inclusion and exclusion of specific atomic features) from FDL to ContextL. Instead, atomic layers can be activated and deactivated just like plain layers in ContextL, either directly, for example by mentioning them in invocations of `with-active-layers` and `with-inactive-layers`, or indirectly via activations and deactivations of composite layers (see below).

Diagram constraints are checked whenever atomic layers are activated or deactivated. If a constraint is not met, an error is signalled at runtime. For example, none of `phone-tariff-a`, `phone-tariff-b` or `phone-tariff-c` can ever be activated unless `phone-call-layer` is already active as well, since `phone-call-layer` is required by all former layers. Likewise, none of the three atomic phone tariff layers can be activated when another one is already active due to the mutual exclusion between those three layers.

Since errors in Common Lisp are resumable [15], signalling them here allows users or programmers to interactively fulfil unmet constraints, and subsequently continue execution of a running program. For that purpose, ContextL provides additional information about a constraint conflict along with the error. For example, a program could present a dialog box like the one in Fig. 2 to allow the user to resolve such a conflict.

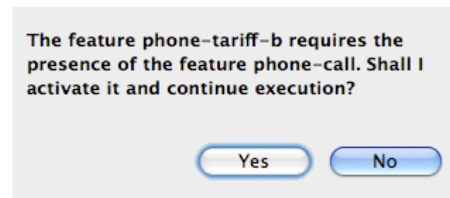

Fig. 2. Interactive resolution of dynamic constraint conflicts.

B. Activation of Composite Layers

Activation of composite layers (via `with-active-composite-layer`, see Sect. III-D) is performed in three steps:

1) The disjunctive normal form of a composite layer's feature expression is determined.
2) The possible feature alternatives described by the DNF are checked against the constraints imposed by both the currently active atomic layers and the atomic layers of each feature alternative.
3) From the remaining set of acceptable feature alternatives, one is selected, and the respective atomic layers are activated, as described above.

The DNF of a composite layer's feature expression can be statically determined, so the result of Step 1 can be cached for each composite layer. Step 2 requires dynamic checks in the general case, since the concrete constraints which need to be checked depend on the dynamic set of currently active layers. Only if these checks unambiguously yield exactly one feature alternative, the results of Step 2 can be cached as well.

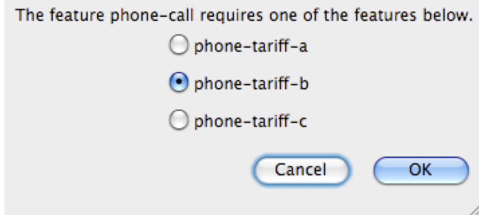

Fig. 3. Interactive selection from multiple feature alternatives.

In Step 3, three cases need to be covered: Either the set of acceptable feature alternatives is empty, contains exactly one alternative, or contains many possible alternatives. In case there are no acceptable feature alternatives, a resumable error is signalled at runtime. In case there is exactly one acceptable feature alternative, it is selected and the respective atomic layers are activated. For example, if Step 1 yields `(one-of (all-of phone-tariff-a))`, and none of the current constraints prevents `phone-tariff-a` from being activated in Step 2, it will indeed be activated in Step 3.

If multiple feature alternatives are acceptable in Step 3, a resumable error signals the ambiguity, and the different acceptable feature alternatives are provided as possible restarts, which can be selected to continue execution of the interrupted program. For example, a program could present a dialog box like the one in Fig. 3.

C. Deactivation of Composite Layers

In feature diagrams, there is no notion of expressing the exclusion of specific composite features, for example as user-defined constraints. Feature diagrams are primarily designed for static selection and composition of software features, not for dynamic activation and deactivation, so the primary purpose of composite features is to enable the positive selection of consistent sets of features.

It could be envisioned that, for example, the exclusion of `phone-tariff` requires the exclusion of *all of* `phone-tariff-a`, `phone-tariff-b` and `phone-tariff-c`, since as soon as *one of* them is active, the feature expression of `phone-tariff` is fulfilled. However, what does it mean to deactivate `internet-tariff`? Is it sufficient if only *one of* the `all-of` branches is not fulfilled? How do we treat the embedded `optional` feature under that circumstance?

The deactivation of a composite layer actually only makes sense if it has been activated before and is thus currently active. This gives us a handle on providing useful semantics for the deactivation of composite layers (via `with-inactive-composite-layer`, Sect. III-D): Whenever a composite layer is *activated* and the atomic layers of a specific, unambiguosly or interactively determined feature alternative are activated as a consequence, that set of atomic layers is recorded for later deactivation of the same composite layer. Recall from Sect. II-A that the activation of an already active layer is ignored. Thus whenever an active composite layer is deactivated, the set of atomic layers selected by an active composite layer is unambiguous for the current dynamic scope.

For example, assume that the composite layer `internet-tariff` is activated and that the feature alternative consisting of the atomic layers `phone-tariff-b` and `flat-rate-1` is selected for activation. When `internet-tariff` is deactivated subsequently, `phone-tariff-b` and `flat-rate-1` will be deactivated as a consequence.

Assume again that `internet-tariff` and thus `phone-tariff-b` and `flat-rate-1` have been selected for activation. Subsequent activation of the composite layer `phone-tariff` leads to activation of the atomic layer `phone-tariff-b`, which is silently ignored (see Sect. II-A).[4] However, we have to record the number of activations of atomic layers, since `phone-tariff-b` is now present due to both `phone-tariff` and `internet-tariff`. This means that whenever only one of those composite layers is deactivated, `phone-tariff-b` must remain active to sustain the other. Only if both composite layers are deactivated, the activation count for `phone-tariff-b` drops to zero, and that layer can thus be deactivated as well as a consequence. Since layers are always active at most once in ContextL, such a simple reference counting scheme is sufficient to keep active composite layers consistent.

D. Implementation

We have implemented feature descriptions for ContextL as a straightforward extension on top of its reflective API. Atomic layers and composite layers are provided as new layer classes. Activation and deactivation of atomic layers require additional checks against constraints, but since for specific sets of active layers, the outcome of such checks is always the same, ContextL's provisions for caching active layer contexts can be exploited [6], [7].

Composite layers are prevented from being activated or deactivated directly. Instead, we have provided two new macros `with-active-composite-layer` and `with-inactive-composite-layer` which perform the three steps for composite layer activation described in Sect. III-B and the deactivation of previously selected

[4] The layers `phone-tariff-a` and `phone-tariff-c` are not acceptable under this circumstance because they are excluded by the presence of the already active `phone-tariff-b`.

atomic layers described in Sect. III-C. The disjunctive normal forms of feature expressions are cached per composite layer. The result of Step 2 of composite layer activation can be cached in case there is exactly one acceptable feature alternative, but we have not implemented such caching yet.

IV. Related Work

Feature-oriented, aspect-oriented and context-oriented programming share the goal to modularize otherwise potentially crosscuting features of a software system, whether they are represented as layers, aspects, or otherwise. In feature-oriented programming, for example, the notion of mixin layers [16] is one approach that allows expressing features as sets of partial class definitions. Such layers can then be selected and composed at compile time. Indeed, the focus of feature-oriented programming lies on static composition and, more recently, on tools for statically analyzing the validity of statically composed systems [1]. However, it has been recently stressed that more dynamic approaches are needed as well [13].

Aspect-oriented technologies approaching the context-oriented notion of dynamically scoped activation of partial program definitions are AspectS [10], LasagneJ [18], CaesarJ [14], and Steamloom [4]. They all add constructs for dynamically scoped activation of partial program definitions, but are limited with regard to when and where such activations can occur, and whether they can be deactivated again. Other aspect-oriented approaches can express conditions under which a given aspect is applicable or not using `if` and `cflow`-style pointcuts, but these conditions can typically only be expressed at the aspect level [17].

V. Summary and Future Work

This paper introduces a declarative extension on top of ContextL's reflective API for describing feature expressions associated with composite layers and constraints between atomic layers. This extension is based on FDL [9], and our integration of FDL retains most of the advantages of ContextL's reflective facility with regard to flexibility and efficiency we have previously described [7].

Future work includes static analysis of feature descriptions for COP, including control flow analysis to predict potential contradictions or ambiguities in feature expressions. As described in [9], FDL allows specification of *default* features in alternative and non-exclusive selections of features to steer the selection of feature alternatives in case of ambiguities. We plan to explore how well this translates to the dynamic activation and deactivation of composite layers. We also need to measure the effect of composite layer activation and deactivation on performance in detail. Especially, each activation and deactivation of a composite layer requires a check against the constraints of currently active layers, which is an NP-complete problem in the general case [2]. We plan to investigate how techniques based on static analysis and partial evaluation, among others, can increase efficiency here.

Acknowledgments We thank Johan Brichau for the initial idea to use feature diagrams in Context-oriented Programming. We thank Charlotte Herzeel, Robert Hirschfeld and Jorge Vallejos for constructive criticism and feedback.

References

[1] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. *Proceedings of the International Conference on Software Engineering 2004*, ACM Press.
[2] D. Batory, D. Benavenides, A. Ruiz-Cortes. Automated Analysis of Feature Models: Challenges Ahead. *Communications of the ACM*, Vol. 49, No. 12, ACM Press, December 2006.
[3] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, D. Moon. Common Lisp Object System Specification. *Lisp and Symbolic Computation* 1, 3-4 (January 1989), 245-394.
[4] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann. Virtual Machine Support for Dynamic Join Points. *AOSD 2004*, Proceedings, ACM Press.
[5] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming. *ACM Dynamic Languages Symposium 2005*. Proceedings, ACM Press.
[6] P. Costanza, R. Hirschfeld, and W. De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. *Joint Modular Languages Conference 2006*, Proceedings, Springer LNCS.
[7] P. Costanza and R. Hirschfeld. Reflective Layer Activation in ContextL. *ACM Symposium on Applied Computing 2007 (SAC 2007)*, Technical Track on Programming for Separation of Concerns (PSC 2007), Proceedings, ACM Press.
[8] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.
[9] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1-17, 2002.
[10] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. *Objects, Components, Architectures, Services, and Applications for a Networked World*, Springer LNCS, 2003.
[11] R. Hirschfeld, P. Costanza, O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, ETH Zurich, 3/4 2008.
[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
[13] J. Lee, K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. *Software Product Lines, 10th International Conference (SPLC 2006)*, Proceedings, IEEE Computer Society.
[14] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. *AOSD 2003*. Proceedings, ACM Press.
[15] Kent Pitman. Condition Handling in the Lisp Language Family. *Advances in Exception Handling Techniques*, Springer LNCS, 2001.
[16] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Design. *ACM Transactions on Software Engineering and Methodology*, March 2002.
[17] E. Tanter. On Dynamically-Scoped Crosscutting Mechanisms. EWAS 2006.
[18] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, B.N. Jorgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. *ICSE 2001*, Proceedings.