Dynamic Contract Layers

Robert Hirschfeld Michael Perscheid Christian Schubert Malte Appeltauer
Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

ABSTRACT

Design by Contract (DBC) is a programming technique to separate contract enforcement from application code. DBC provides information about the applicability of methods and helps to narrow down the search space in case of a software failure. However, most DBC implementations suffer from inflexibility: Contract enforcement can only be activated or deactivated at compile-time or start-up, contracts are checked globally and cannot be restricted in their scope such as to the current thread of execution, and contracts cannot be grouped according to the concerns they relate to.

In this paper, we present *dynamic contract layers (DCL)* for fine-grained and flexible contract management. Based on ideas from context-oriented programming, we extend DBC by a grouping mechanism for contracts, thread-local activation and deactivation of such groups, and selective contract enforcement at run-time. *PyDCL*, our proof-of-concept implementation of DCL, is built onto *ContextPy*, our COP extension for the Python programming language. We evaluate our approach by applying PyDCL contracts to the MoinMoin Wiki framework.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Object-Oriented Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

design by contract, dynamic contract layers, context-oriented programming, software composition

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'10 March 22-26, 2010, Sierre, Switzerland.
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...$10.00.

1. INTRODUCTION

Design by Contract. With *Design by Contract* (DBC [16, 21]) programmers can separate the specification of pre- and postconditions and invariants from their application code to improve code comprehensibility and quality. Such contracts are explicit, machine-readable, and executable.

Preconditions check properties that must hold whenever a method is entered. For example, preconditions can assert that the object a method is invoked on is in a valid and expected state and that the parameters passed to this method are both of the right type and conform to the constraints imposed by the application domain.

Postconditions check properties that must hold whenever a method returns to its caller. For example, postconditions could check both if the object to be returned by a method invocation conforms to constraints from the application domain and if the object the method was invoked on is left in the expected state.

Invariants are defined for an entire class of objects and are to be preserved by all invocations of (public) methods.

In DBC, pre- and postconditions and invariants form a *contract* that—together with all (public) method signatures—is part of the specification of the class. Such contracts are established between the clients (or users) and the supplier (or implementor) of a class. When calling a method, the client has to make sure that all preconditions of this method are fulfilled, while the supplier must assert that the method returns a result that complies to the postconditions and leaves the method's object in a state that satisfies its invariants.

DBC Limitations. The ideas of DBC [21] and their first incarnation in the Eiffel programming language [22] have contributed to software engineering for almost two decades now [27]. Still, for many applications, this approach is too inflexible. In the following, we list some of the limitations of DBC we are going to address in this paper.

Grouping. Pre- and postconditions are bound to a method and invariants are bound to a class. There is no way to explicitly group contracts or parts thereof. They are only grouped implicitly by the given class structure. For instance, grouping contracts related to specific requirements is almost impossible since the implementation of most of the requirements are spread over several classes and so cannot be related to a single one [7, 9].

Scoping. Contracts can be activated or deactivated globally and only for an entire compilation unit. This can be

an issue in a multi-threaded multi-user environment such as a Web server, where requests from different users are processed in different threads. Enabling contract enforcement on a per-thread basis would, for example, allow developers and maintainers to limit the scope of these contracts to a particular control flow while the rest of the system remains unaffected.

Dynamic Enforcement. In Eiffel's original implementation and many other approaches to DBC, contracts are static properties that are activated or deactivated at compile-time. The en- and disabling of contracts in such systems can be difficult, since it requires that executables with or without contracts have to be re-compiled, binaries in the production system must be exchanged, and the system has to be restarted.

Contributions. In this paper, we present *dynamic contract layers (DCL)*, which augment the flexibility of state-of-the-art DBC approaches by combining them with ideas from context-oriented programming (COP [14]). DCL provides a contract grouping mechanism orthogonal to or crosscutting class hierarchies. It allows for a fine-grained thread-local contract enforcement, where contracts can be selectively activated and deactivated at run-time.

PyDCL is our implementation of DCL in and for the Python programming language [28]. It builds on top of and extends *ContextPy*, our library-based COP-extension to Python. We evaluate DCL by applying PyDCL contracts to the MoinMoin Wiki framework [11].

Outline. The remainder of this paper is organized as follows: Section 2 gives an overview of context-oriented programming (COP) our approach is based on. Section 3 describes how we leverage the concepts of COP for DBC and especially DCL. Section 4 explains our implementation of ContextPy and PyDCL in more detail. Section 5 discusses preliminary findings. Section 6 presents related work. Section 7 concludes the paper and suggests future work.

2. CONTEXT-ORIENTED PROGRAMMING

We propose an approach to dynamically compose contracts to overcome the limitations of DBC introduced in the previous section. It turns out that context-oriented programming [14] (COP) is an ideal base for our extension to DBC. For that reason, we give a brief introduction to the COP paradigm. A broader introduction to COP is provided in [14]. In the next section, we describe how we leverage the concepts to enable grouping of contracts, run-time activation, and thread-local contract checking.

2.1 Overview

COP supports the explicit representation of context-dependent behavioral variations. Such behavioral variations can be grouped for comprehensibility and programmer convenience. They are activated or deactivated dynamically at run-time, depending on the context of execution. For safety reasons, most COP implementations allow to limit such compositions to the local thread of control and to scope their effect to the dynamic extend of a specific section of the program. So far, COP has been implemented for several languages such as Lisp [5], Smalltalk [13], Python [29], Ruby [26], and Java [2]. An overview of most of them is provided in [1].

```
1  employerLayer = layer("Employer")
2
3  class Person(object):
4      def __init__(self, name, employer):
5          self.name = name
6          self.employer = employer
7
8      @base
9      def getDetails(self):
10         return self.name
11
12     @around(employerLayer)
13     def getDetails(self):
14         return proceed() + "\n" + self.employer
15
16 person = Person("Michael Perscheid", "HPI")
17 print person.getDetails()
18 with activelayer(employerLayer):
19     print person.getDetails()
```

Listing 1: ContextPy Example

Context. Context is everything that can be accessed from within a program, ranging from the domain information it represents and the system's meta-structure including its control flow to external state or events. *Behavioral variations.* Behavioral variations are partial methods that can be executed around, before, or after the original functionality. During its execution, a partial method can *proceed* to the next variation provided by another layer or, if no such variation exists, to the next base method. *Modularization.* COP introduces *layers* to group behavioral variations. These variations can crosscut the underlying module structure such as the application's class hierarchy. Layers group *partial method definitions* implementing behavioral variations. *Dynamic composition.* Layers can be activated or deactivated from within the program at run-time. *Scoping.* Layer activation is controlled on a per-thread basis. In most implementations so far, it is also scoped to the dynamic extent of the execution of a block of statements.

2.2 ContextPy

Listing 1 presents an illustrative example written in *ContextPy*, our context-oriented extension to Python.

First, we create a layer object `employerLayer` (Line 1). After that, we implement a simple `Person` class with two attributes describing the person's `name` and `employer` (Lines 3–14). The first definition of `getDetails` (Lines 8–10) represents the base method definition, as marked by the `@base` decorator. It implements the default behavior returning the person's name attribute. The second method (Lines 12–14) is a partial definition of `getDetails` and annotated with the `@around` decorator. This method will be executed instead of the base method whenever the layer `employerLayer` is active. By using the built-in method `proceed` within the method body (Line 14), we invoke the next partial or base method.

After defining this class, we instantiate a `Person` object and call `getDetails` twice, first without, and then with layer activation (Lines 16-19). The execution without any layer (Line 17) directly returns the `name` attribute. The second invocation is placed within a `with` block that activates `employerLayer`. Here, the method invocation also returns information on `employer` in addition to the person's `name`.

```
1 from MoinMoin.Page import Page
2 from pydcl import requireContract, ensureContract
3 import cop, copy
4
5 textProcessing = cop.layer("Text Processing")
6
7 class PageEditor(Page):
8
9   @requireContract(__layer__ = (textProcessing,))
10  def normalizeText(self, text, **kw):
11    assert isinstance(text, (str, unicode))
12    assert kw.get('stripspaces', 0) in (0, 1)
13
14  @ensureContract(__layer__ = (textProcessing,))
15  def normalizeText(self, text, **kw):
16    oldDict = {"lock": copy.deepcopy(self.lock)}
17    result = proceed(text, **kw)
18    assert isinstance(result, (str, unicode))
19    assert self.lock is oldDict["lock"]
20    return result
21
22  def normalizeText(self, text, **kw):
23    if text:
24      lines = text.splitlines()
25      if kw.get('stripspaces', 0):
26        lines = [line.rstrip() for line in lines]
27      if not lines[-1] == u'':
28        lines.append(u'')
29      text = u'\n'.join(lines)
30    return text
```

Listing 2: DCL Pre- and Postconditions

3. DYNAMIC CONTRACT LAYERS

In this section we present *Dynamic Contract Layers* (DCL), our extension to DBC. We describe our COP-based implementation of DCL, its improvementsover DBC, and a domain-specific language for DCL contracts. We apply DCL to the *MoinMoin Wiki* [11] taking MoinMoin's normalization of user input that is part of the text processing feature as an example. Within the page editor, users can insert text to change the content of a wiki page. Such text must be normalized before it is stored by MoinMoin since different browsers and operating systems may create different input values.

3.1 DCL Contracts

As in the original DBC, DCL Contracts consist of pre- and postconditions and invariants.

Preconditions ensure that the input values have the right properties and that the object is in a valid state. The mapping of preconditions to COP concepts can be expressed straightforwardly by partial methods that are executed before the base method. Developers implement their assertions via partial methods (with the same signature as the base method) and define it as a precondition. We provide the annotation `requireContract` for that. In addition to explicitly defined layers, we also add the globally accessible `requireLayer` to each method. Listing 2 presents our example from MoinMoin. Lines 9-12 define a partial method for preconditions. The `__layer__` parameter (Line 9) declares an additional layer to group this condition. When the `requireLayer` or `textProcessing` layer is activated, the precondition method is executed before `normalizeText`, which asserts that the input parameter `text` has the expected type and the keyword dictionary is properly filled.

Postconditions ensure that the return values have the right properties and that the object the message was sent to is still in a valid sate after returning to its caller. Since DBC requires the comparison of *old* values—state before the method was executed—with newly computed values after the execution, a simple mapping to partial methods being executed after the original one is not sufficient. Thus, we map postconditions to around methods. With that, developers can store old values, proceed to the original method, get the return values, and implement the postconditions. Similarly to the preconditions, we have specified a new annotation called `ensureContract` that automatically adds the `ensureLayer`. Listing 2 (Lines 14-20) illustrates postconditions within MoinMoin. However, this approach has also some drawbacks—the need to explicitly store old values, to explicitly call proceed, and to explicit pass on the result. Here COP boilerplate code is in the way of the developers and with that a source of errors. In Section 3.5, we address these problems by introducing a DCL contract definition language.

```
@invariant()
def attributeTypes(self):
 assert isinstance(self.do_revision_backup, (bool,int))
 assert isinstance(self.do_editor_backup, (bool,int))
 assert isinstance(self.lock, PageLock)
 assert isinstance(self.uid_override, (NoneType,str))
```

Listing 3: A DCL Invariant

In DBC, *invariants* belong to a class and ensure that some properties of that class are ensured at all time. An invariant is usually evaluated before and after every call to a (public) method. We map an invariant to two partial methods—one executed *before* and one executed *after* each method. We introduce an `invariant` annotation inserting an `invariantLayer` for each partial method. Furthermore, invariants must be implemented with respect to the *assertion evaluation rule* (AER) of DBC. During the evaluation of an assertion all subsequent method calls must be executed without any evaluation [23]. Thus, we have to ensure that all layers are deactivated during the execution of our partial methods. We have extended COP with an all-layers-deactivation that is wrapped around our invariants—a `with` clause executes the invariant in the dynamic extend of all deactivated layers, otherwise resulting in an infinite recursion. Listing 3 shows an invariant for the `PageEditor` class of MoinMoin that checks the proper types of each attribute within that class.

3.2 Contract Grouping

Contract grouping and selective contract group activation and deactivation allow us to keep things together that belong together. Criteria for forming such groups are up to the developers, but requirements representations such as use cases oder user stories work nicely for that. For example, a contract group can encapsulate distinct application features or certain kinds of checks, such as correct typing or the range of parameter values. Also, instead of checking the entire system, developers can provide contract groups to verify system properties that most likely have to do with the cause of the observed problem. Whenever necessary, developers can activate or deactivate one, many, or all such groups.

We employ layers to group contracts decoupled from the parts of class hierarchy they apply to. In COP, each layered

method is assigned to exactly one layer, thus, each contract layer would only belong to one group. DCL contracts should, however, belong to several such groups. For that reason, we have extended our approach so that partial methods can be associated with more than one layer. Here a partial method will be executed if at least one of the layers it is associated with is activated.

3.3 Contract Scoping

Most DBC frameworks support only global contract activation. However, enabling all contracts may impose a significant performance overhead. For instance, if we want to trace an error in a multi-threaded server application by enforcing contracts, we would have to activate all contracts for the entire system, which can lead to a decrease in performance.

In DCL, *thread-based contract scoping* restricts contract enforcement to a specific control flow. In a production system, for example, developers test system behavior without influencing the execution flow of concurrent clients. This can especially be useful in multi-threaded Web applications where developers want to debug the system by checking contracts deployed into the hosted system without disrupting uninvolved users or with decreasing performance. To only check contracts of classes and methods involved in a specific control flow and a selected group of contracts, DCL takes advantage of dynamic layer composition via COP's `with` statement.

In addition to *dynamic extent-based scoping* of the basic COP approach, we have extended our ContextPy implementation with the feature of *global layer activation*. In general, dynamic adaptation must ensure the system's consistency, thus, global activation should be carefully applied. However, DCL's contracts are not expected to affect the behavior of any base method, so global activation of dynamic contract layers should be applicable without the introduction of modifications to the system's original behavior.

3.4 Dynamic Contract Enforcement

Classic DBC contract enforcement being a compile-time or startup option requires to restart or even to recompile a deployed system in order to switch contracts on or off, which obviously is impractical for, for example, running diagnostics in a production setting and supporting short feedback cycles in an agile development environment.

DCL's *dynamic contract enforcement* deals with the activation and deactivation of contracts at run-time. Dynamic composition is a basic property of COP and since DCL is based on COP, it inherits this property automatically. With the help of COP's `with` statement, developers can choose which contracts to be activated where and when, without the need to rebuild the complete system. The activation of corresponding contract layers can be done directly by the developers in the code or by some external means such as a debugger or other development or deployment tools.

3.5 A Domain-specific Language for DCL

In the previous examples, we presented DCL using plain COP language support. Declaring contracts using partial methods is a powerful means, since these partial methods can contain any code that might be necessary to check the contracts. However, for most applications of DCL such as type constraints and simple assertions, this expressiveness is not necessary but may lead to longer and more complex contract definitions instead. Here, we provide a very small domain-specific language that can be used directly before a method definition. So, developers can both write and see the contracts next to the methods they apply to. We offer two annotations—`@require` and `@ensure`—for the definition of simple pre- and postconditions,. The assertion statements and the optional list of contract groups are passed as arguments to these annotations.

Some postconditions require to compare an object's state before and after a method execution. Using ContextPy syntax, we create a dictionary to which we put a deep copy of the object, call `proceed` to execute the method, and compare the new state with the copied one. For this idiom, our DSL provides a build-in method called `old` that refers to an object's state before the method was executed. Furthermore, developers can access the return value with the special keyword `__result__`.

In addition to that, our DSL provides two annotations called `@requireTypes` and `@ensureTypes` to state type constraints for parameters and return values. This is particularly helpful in dynamically typed languages. As arguments, the first annotation accepts the method's parameters with assigned lists of allowed types and the second one expects the possible return types. Both annotations can be extended with additional contract layers.

Listing 4 shows a refactoring of the example introduced in Listing 2 using this domain-specific language.

4. IMPLEMENTATION

In this section, we present the implementation of *ContextPy*, our COP library for Python, and *PyDCL*, its DCL extension.

4.1 ContextPy

ContextPy is our COP implementation for the Python programming language[1]. It supports the layer-in-class approach [1] and with that allows developers to define their partial methods within the scope of the actual classes these methods are contributing to. Similar to all other COP extensions so far, ContextPy provides both layers, partial methods, and dynamic scoping.

Layers. In ContextPy layers are represented by regular objects that provide the identities layers need to exhibit at run-time. Layer access has to be managed by the developers.

Layer Composition. The representation of activated and deactivated layers is handled by two stacks—one for thread-specific and one for globally activated layers. Layers can be (de-)activated using Python's `with` statement or by library methods for stack operations.

Partial Methods. In Python, each class has its own dictionary that maps identifiers (keys) to objects (values). For instance, a method is stored with its name as key and the method object as value. We introduce a layered method descriptor that consists of all (partial) methods and a cache

[1]PyContext [29] is another COP library in Python. Its class-in-layer approach is not suitable for DBC since contracts should be located close to the base method they apply to.

```
1 from MoinMoin.Page import Page
2 from pydcl import require, requireTypes, ensure, ensureTypes
3
4 textProcessing = cop.layer("Text Processing")
5
6 class PageEditor(Page):
7
8   @require("kw.get('stripspaces',0) in (0,1)", __layer__ = (textProcessing,))
9   @requireTypes(text = [str, unicode], kw = [dict], __layer__ = (textProcessing,))
10  @ensure("old(self.lock) == self.lock", __layer__ = (textProcessing,))
11  @ensureTypes(str, unicode, __layer__ = (textProcessing,))
12  def normalizeText(self, text, **kw):
13    ...
```

Listing 4: Simplified DCL Contract Definition

to store behavioral variations. As soon as a partial method is found in Python's class initialization, we change the base method object to our layered method descriptor and store all further partial methods, the related layer, and the kind of execution within this descriptor object.

Layer-aware Method Lookup. We are leveraging Python's binding mechanism so that—to developers—there is no difference between a call to a base method and a layered method descriptor. During such a call, we are creating a proxy object that computes which partial methods should be executed based on the activated layers pushed onto both layer stacks. More precisely, we are creating a linked list of so-called advice consisting of the partial method object and a reference to the next method in the list. This is done only once and then stored in the cache of the layered method descriptor. By calling the advice chain, each advice decides to forward the call or not depending on its type and the proceed statement.

4.2 PyDCL

PyDCL implements DCL based on ContextPy.

Design by Contract. As emphasized previously, implementing pre- and postconditions as pure partial methods is straightforward. However, the realization of invariants requires a new meta class that overrides the initialization of classes. If an invariant is detected during the class construction, we store this partial method in a separate list and change the meta class. Having all methods in the class dictionary, the overridden initialization is called which inserts all invariants before and after each method. Thus, we transform each method into a layered method descriptor with the corresponding invariants as partial methods. The AER principle is realized by an additional decorator around each invariant that disables all layers before and activates them again afterwards.

Domain-specific Language. The two contract definition languages are implemented in three steps. In the analysis step, we identify the method signature of the original method and the value of the optional `__layer__` parameter. In the transformation step, we translate the input of the contract definition languages into Python source code. Direct conditions are translated into pure assertions and type contracts are further wrapped by an `isinstance` method. The implementation of the `old` keyword creates additional source code for postconditions; annotated objects are deeply copied before the proceed call to the original method happens and afterwards inserted in the related assertions. In the compile step, the generated source code is compiled in anonymous partial methods. These methods and the original one are added to a layered method descriptor.

5. DISCUSSION

In the following, we discuss what we have learned from the implementation of our approach to DCL and its application to the MoinMoin Wiki.

Contract Definitions. Writing contracts for each method manually is a laborious and tedious task. Often, developers do not even know which pre- and postconditions a method has to fulfill or which invariants are meaningful for a class. In future work, we are interested in exploring inductive debugging concepts in combination with automatically generated contracts [32]. This knowledge should be derived from runtime data of controlled and passed unit and acceptance tests. For instance, the parameter of a method has an integer type in each observed run. It is very likely that the method always expects an integer as input type and, based on this assumption, we can establish a corresponding precondition.

Layer Assignment. We suggest features as primary grouping concept from the user's point of view. Most often, failures are discovered by users. And usually the users describe the erroneous behavior of the system from their perspective. Thus, it makes sense to choose and activate corresponding feature layers that check contracts of participating methods. Unfortunately, so far developers have to identify these feature-method-relationships manually. Feature analysis [9] is a research area that provides an answer to the question which methods are concerned in which features. By combining feature analysis with our approach to DBC, we hope to automate the process of feature-based grouping contracts within layers.

Evaluation. Our experiments with MoinMoin are a first step to validating our approach. More case studies are required to evaluate DCL in a more elaborate manner. First and foremost, we are interested in developer feedback to improve our approach for larger-scale projects.

6. RELATED WORK

The roots of DBC can be traced back to the work of Floyd [8] and Hoare [15]. Later work by Parnas [24] already

shows ideas to be found in DBC. Besides the original DBC implementation in Eiffel [21], we can find DBC extensions to several other programming languages.

Today, there are at least three freely available implementations of DBC in Python: Design by Contract by Plösch [25], *ContractPy* [31], and *PyDBC* [4]. While PyDBC—much like our work—uses seperate (partial) methods for checking preconditions, postconditions, and invariants and using a wrapper method to invoke partial methods, it has no dynamic properties and no support for the AER. ContractPy, like the implementation by Plösch, is based on *docstring*-parsing. For every module or class an explicit function call is required to parse these docstrings, compile the assertions, and replace the methods by wrappers that perform the checks. While the docstring approach provides a tighter notation, they are usually limited in functionality. There are no provisions for avoiding recursive contract checks, but the `old` notation is supported.

There are even more implementations for Java. *iContract* by Kramer [20] uses structured comments (like [25, 31]) for specifying contracts and provides a preprocessor that instruments the original source code with contract checks. It has sufficient support for the AER to avoid recursive contract checks. There is no support for the `old` notation. *jContractor* by Karaorman et.al. [17] places contract checks in separate methods. A factory method or alternatively a class loader transforms these methods (apparently on a byte-code level) to invoke the contract checks. There is support for the `old` notation, but there does not seem to be provisions against recursive contract checks. In Duncan and Hölzle's *Handshake* [6] contracts are read from a separate file with a Java-like syntax. Those contracts can be compiled separately. It comes with a modified version of the standard C library which is used by the JVM for file I/O. Handshake intercepts such calls and supplements the class files with contracts. It does not depend on a class-loader-base mechanism (which might not be available in all Java implementations), nor does it require modifications to the code (neither the client nor the supplier). Since the classes are modified on a byte-code level and the contracts can be written without access to the source code. As a downside, their implementation neither supports the `old` notation, nor does it prevent recursive assertion evaluation.

Guerreiro [10] implements contracts for C++. Classes which want to use contracts, need to derive from a special class that provides this functionality. There is limited support for the old notation which requires the explicit recording of the pre-state. He claims that this approach supports activation and deactivation of contract checks per class and at run-time. However, there is no lazy evaluation in C++, assertions will be evaluated in any case and the result is disregarded when the checks are disabled. Errors are hidden and performance will not increase significantly. The approach does not avoid recursive assertion checks.

Aspect-oriented programming [19] (AOP) provides language abstractions to encapsulate crosscutting concerns, thus it is conceptually related to COP. In general, AOP separates the implementation of crosscutting concerns from the main application logic. In contrast to that, COP allows for the specification of partial methods close to their base definitions. This declaration style corresponds to contract specifications next to their respective methods. The specific features of DCL such as contract grouping, contract scoping, and dynamic contract activation can be represented by COP abstractions more naturally than with aspects.

Contract4J [30] is an *AspectJ* [18] based tool that supports classic DBC. As in our PyDCL, contracts are specified via annotations. However, Contract4J is not as dynamic as PyDCL since AspectJ does not support dynamic aspect deployment. Some AOP languages, such as CaesarJ [3] and AspectS [12] overcome this limitation and provide dynamic aspect weaving, though they lack in supporting annotation-based pointcuts. Since our goal is to specify contracts close to their methods, a pointcut-based description within an aspect module appears undesirable.

7. SUMMARY AND OUTLOOK

In this paper, we have presented *dynamic contract layers*, or DCL, as an extension to design by contract. Based on context-oriented programming, its layer construct, and its dynamic means of composition, we extend the flexibility of current design by contract approaches in three ways. First, DCL allows for expressing contracts as partial method definitions and for grouping them into contract layers, preferably according to the concerns a particular subset of contracts has to support. Second, DCL enables the scoping of contract layer activations to the local thread of execution. Third, DCL facilitates contract layer activation and deactivation dynamically at run-time.

Future work will be concerned with inductive debugging techniques [32]. We want to investigate automated contract generation. Using dynamic analysis techniques, we would like to process trace data from unit tests to derive from that sound assertions covering both state and behavior. We are also interested in automatically relating contract layers to requirements and other artifacts throughout the entire software lifecycle.

Acknowledgments

We would like to thank Gregor Schmidt, Martin von Löwis, and Michael Haupt for fruitful discussions and valuable contributions.

8. REFERENCES

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-Oriented Programming Languages. In *International Workshop on Context-Oriented Programming*, pages 1–6, 2009.

[2] M. Appeltauer, R. Hirschfeld, and H. Masuhara. Improving the Development of Context-dependent Java Applications with ContextJ. In *International Workshop on Context-Oriented Programming*, pages 1–5, 2009.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, pages 135–173, 2006.

[4] D. Arbuckle. PyDBC - Contracts for Python 2.2+. Version 0.2. `http://savannah.nongnu.org/projects/pydbc/`.

[5] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Dynamic Languages Symposium*, pages 1–10, 2005.

[6] A. Duncan and U. Hoelzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, USA, 1998.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. In *Transactions on Software Engineering*, volume 29, pages 210–224, 2003.

[8] R. W. Floyd. Assigning Means To Programs. In *Mathematical Aspects of Computer Science*, volume 19, pages 19–31, 1967.

[9] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Bern, 2007.

[10] P. Guerreiro. Simple Support for Design by Contract in C++. In *Technology of Object-Oriented Languages and Systems*, page 24, 2001.

[11] J. Hermann. The MoinMoin Wiki Engine - Easy to Use, Full-Featured and Extensible Wiki Software. Version 1.8.4. `http://moinmo.in/`.

[12] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In *International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, 2003.

[13] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-Oriented Programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407, 2008.

[14] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. In *Journal of Object Technology*, volume 7, pages 125–151, 2008.

[15] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. In *Communications of the ACM*, volume 26, pages 53–56, 1983.

[16] J.-M. Jézéquel and B. Meyer. Design by Contract: The Lessons of Ariane. In *Computer*, volume 30, pages 129–130, 1997.

[17] M. Karaorman, U. Hölzle, and J. Bruno. jContractor: A Reflective Java Library to Support Design By Contract. Technical Report TRCS98-31, University of California, Santa Barbara, USA, 1998.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327–354, 2001.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[20] R. Kramer. iContract - The Java Design by Contract Tool. In *Technology of Object-Oriented Languages and Systems*, page 295, 1998.

[21] B. Meyer. Applying "Design by Contract". In *Computer*, volume 25, pages 40–51, 1992.

[22] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.

[24] D. L. Parnas. A Technique for Software Module Specification with Examples. In *Communications of the ACM*, volume 15, pages 330–336, 1972.

[25] R. Plösch. Design By Contract for Python. *Asia Pacific and International Software Engineering Conference.*, pages 213–219, 1997.

[26] G. Schmidt. ContextR & ContextWiki - Modularisierung von Webanwendungen mit kontextorientierter Programmierung. Master's thesis, Hasso-Plattner-Institut an der Universität Potsdam, 2008.

[27] J. Tantivongsathaporn and D. Stearns. An Experience With Design by Contract. In *Asia Pacific Software Engineering Conference*, pages 335–341, 2006.

[28] G. Van Rossum. *The Python Language Reference Manual*. Network Theory Ltd., 2003.

[29] M. von Löwis, M. Denker, and O. Nierstrasz. Context-Oriented Programming: Beyond Layers. In *International Conference on Dynamic Languages*, pages 143–156, 2007.

[30] D. Wampler. Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, 2006.

[31] T. Way. ContractPy - Programming by Contract for Python. Version 1.4. `http://www.wayforward.net/pycontract/`.

[32] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.