Computational Contracts

Christophe Scholliers
Software Languages Lab
Vrije Universiteit Brussel
cfscholl@vub.ac.be

Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
etanter@dcc.uchile.cl

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
wdmeuter@vub.ac.be

Abstract

Pre/post contracts for higher-order functions, as proposed by Findler and Felleisen and provided in Racket, allow run-time verification and blame assignment of higher-order functions. However these contracts treat contracted functions as black boxes, allowing verification of only input and output. It turns out that many interesting concerns about the behaviour of a function require going beyond that black-box approach, in order to control the actual computation that follows from a function. Examples are prohibiting or verifying that certain functions are called, checking access permissions, time or memory constraints, interaction protocols, etc. To address this need for grey-box verification, while preserving support for higher-order programming and blame assignment, we introduce the notion of *computational contracts*. A computational contract is a contract over the execution of a contracted entity.

We show various applications of computational contracts, and explain how to assign blame in case of a violation. Computational contracts have been integrated with the existing contract system of Racket. Computational contracts is the first contract model with blame assignment in a higher-order setting that provides a systematic way to perform grey box verification.

1. Introduction

Design by contract [35] is a software correctness methodology that is based on the principle of pre- and post-conditions to verify certain functionalities of the program. Design by contract was first popularised in the Eiffel [34] programming language and since then adopted in many languages including C, C++, Smalltalk, Haskell, Perl, Python, .NET4 and Scheme. Contracts have also been applied in multithreaded object-oriented systems to coordinate groups of objects [22]. Design by contract is currently the most requested feature[1] to be added to the Java language. Languages with support for higher-order functions can not make use of traditional assertion based contracts. Findler and Felleisen [12] have adapted assertion based contracts to support higher-order functions.

In general the aim of contracts is to specify and verify well-defined properties of a system. Beugnard et al [2] categorise contract systems in four levels: syntactic (type systems), behavioural contracts (pre/post conditions), synchronisation contracts (dependencies between the provided services) and quality of service contracts (e.g. time and space guarantees). We have found that contract systems do not provide a general mechanism to verify the contracted entity *during its execution* in a higher-order setting. We propose a new mechanism that allows the programmer to define and verify contracts over a function (and its possibly higher-order arguments) while it is executing.

A very simple example of the lack in expressiveness of current contract systems can be observed when implementing a contract that disallows a function to write to a file. In current systems it requires the programmer to manually save the state of the file in the precondition and later verify that the state of the file has not changed in the postcondition. While this functionality on its own requires a substantial amount of work, other functions might open, write, and close files in the system *concurrently*. These writes are possibly allowed and thus should be ignored when verifying the postcondition. Implementing this kind of functionality with current contract frameworks is very difficult. Even more importantly, the verification of this particular contract in the (post-condition) is too late i.e. the damage has already been done. Moreover current contract systems do not allow the programmer to detect that a file was read.

The core problem of current contract systems is that they treat a contracted entity as a black box. Many aspects such as prohibiting or enforcing certain method invocations, access permission, time constraints, sending messages over the network, memory constraints etc. are well-defined properties of the computation of a certain function. However current contracts do not provide a structured and expressive mechanism to verify these aspects in a higher-order setting. In this paper we present *computational contracts*, an extension to the higher-order contract systems defined by Findler and Felleisen [12]. Computational contracts tackle the problems of current higher-order contract systems by also allowing grey box verification of the contracted functions. We present various examples of computational contracts including mandatory function calls and protocol contracts. We describe an expressive model to specify and verify higher-order computational contracts, including proper blame assignment. We have implemented[2] computational contracts in Racket and integrated them with the existing contract system.

We start our explanation of computational contracts by first giving a small overview of Racket contracts, in Section 2. Then we contrast the Racket contracts with computational contracts, in Section 3. Examples of computational contract applications are shown in Section 4. Subsequently we explain how computational contracts are verified and how blame is assigned in Section 5. We discuss interactions between computational contracts and existing contracts in Section 6. Before concluding we survey related work in Section 7.

2. Racket Contracts in a Nutshell

The Racket contract system is based upon Findler and Felleisen [12] seminal work on higher-order pre/post contracts. It differentiates between contracts defined over simple values, called flat contracts, and contracts defined over functions dubbed functional contracts. Functional contracts are of the form $C_d \rightarrow C_r$ where C_d is a

[1] `http://bugs.sun.com/bugdatabase/top25_rfes.do`

[2] Available at `http://tinyurl.com/5rzdzoo`.

```
1 (define/contract sqrt
2   (greater-than-zero? .→. greater-than-zero?)
3   (λ(x) ...))
```

Figure 1. Simple contract over the sqrt function.

contract over the domain of the function and C_r is a contract defined over the range of the function. As the Racket contract system supports higher-order pre/post contracts it allows C_d and C_r to be either flat or functional contracts. We first show an example where C_d and C_r are flat contracts followed by an example where C_d is a functional contract. Finally we show an extension to higher-order pre/post contracts that is expressive enough to verify certain quality of service (QoS) contracts.

2.1 First-Order Function Contracts

The prototypical example of contract frameworks is that of a contracted `sqrt` function. The purpose of the contract is to ensure that the argument passed to the `sqrt` function is a positive number (precondition) and that the result of the `sqrt` function is also a positive number (post-condition). A possible specification of this contract is shown in Figure 1. In the example there is one functional contract defined over the function `sqrt` that is itself composed out of two flat contracts. The flat contract over the argument of the `sqrt` function `greater-than-zero?` verifies that the arguments passed to the `sqrt` function is positive (left from the arrow). Similarly the result of the function (after the arrow) must also pass the same flat contract.

Programmers using contracts do so for two main reasons: documentation of the interfaces and to enforce the contract by letting the contract system point out the responsible party in case of a violation. The process of figuring out who violated the contract is called blame assignment [12]. For a long time researchers agreed that assigning blame was simply a matter of determining where the violation took place. If the pre-conditions are violated it is the callers fault, if the post-condition is violated it is the callee's fault (the `sqrt` function in our example). However in the context of languages such as Ruby, Python or Scheme the use of higher-order functions makes blame assignment more challenging.

2.2 Higher-Order Pre/Post Contracts

Higher-order pre/post contracts [12] were first introduced by Findler and Felleisen. In their work they extend traditional contracts so that they can be used in the context of higher-order functions. The key idea behind blame assignment in the context of higher-order functions is to postpone blame assignment of a functional argument until that argument is applied. The reason for this increased complexity is that in the general case it is not possible to verify the contract defined over a functional argument in the preconditions [36]. A simple higher-order pre/post contract function is shown in Figure 2. The `map-pos` function expects a (positive) function `f` and a list `l` and applies the traditional `map` function to the function and the list. The contract defined over the function `map-pos` specifies that the first argument must be a function that expects a value greater than zero and returns a value greater than zero. The second argument should be a list of integers, denoted by `(listof greater-than-zero?)`. Last, the contract specifies that the return value of the function `map-pos` has to be a list of positive integers.

During the execution of this function blame can be assigned either to the caller of the `map-pos` or to the `map-pos` function itself. In case that the function `map-pos` returns a value that is not a list or a list that contains an element that is not a positive number, blame is assigned to the `map-pos` function. Suppose module M_1

```
1 (define/contract map-pos
2   ((greater-than-zero? .→. greater-than-zero?)
3    (listof greater-than-zero?)
4    .→.
5    (listof greater-than-zero?))
6   (λ (f l)  ... ))
```

Figure 2. Higher-order pre/post contract over the function map-pos.

```
1 (define/contract sqrt
2   (greater-than-zero?
3   . →d .
4   (λ (arg)
5      (λ (res)
6         (and (greater-than-zero? res)
7              (<= (abs (- arg (* res res)))
8                  0.01))))))
```

Figure 3. Sqrt contract, refined with a dependent contract.

imports the `map-pos` function from module M_2. If M_1 applies it with a function that does not obey the contract, blame is assigned to module M_1. This is in contrast to naive assertion-based systems, which would assign blame to M_2, because the contract violation was detected within the body of `map-pos`.

2.3 Dependent Contracts

Many contracts need to access some information in the postconditions which is only available when the contracted function is applied. As this pattern is often seen in the construction of pre/-post contracts, an extension called *dependent contracts* was proposed [12] . Dependent contracts allow the argument values of a contracted function to be captured when the contracted function is applied in such a manner that it is accessible in the postcondition of the contract [3]. The canonical example of a dependent contract is a refinement of the `sqrt` contract. On top of verifying that the result of the `sqrt` function is positive it also verifies that the square of the result is within a certain bound of the argument. The refinement of the `sqrt` contract with a dependent contract is shown in Figure 3. As can be seen in the contract, the postcondition of a dependent contract is a lambda expression that receives the argument of the contracted function and returns a regular postcondition contract. In the postcondition the argument value is exposed and thus can be used to verify that the square of the result of the `sqrt` is within 0.01 of the argument.

With dependent contracts we finish the quick overview of the higher-order pre/post contract system in Racket. As can be observed from the work on dependent contracts there is a need for abstraction mechanisms that go beyond simple pre/post contracts. In the next section, we present our extension to higher-order pre/-post contracts, which allows the specification of contracts over the actual computation of a function.

3. Computational Contracts: a First Look

A computational contract is a contract over the execution of a contracted entity. In contrast to existing contracts, which treat a contracted entity as a black box, a computational contract can verify well-defined execution points *during* the execution of the contracted entity. Programmers can specify what has to happen or what

[3] Because dependent contracts can capture arbitrary state they can be misused to implement certain QoS contracts. However, this is not common practice and the default resulting error messages are cumbersome for QoS contracts.

```
1 (define/contract sqrt
2   (greater-than-zero?
3     . -!call(display)-> .
4     greater-than-zero?)
5   (λ(x) ...))
```

Figure 4. Example computational contract.

should not happen during the execution of the contracted entity. Pre/post contracts do not allow the programmer to express concerns like,"does the contracted entity reads a file","does it send data over the network", etc. With computational contracts we allow the programmer to express these concerns. Depending on the level of expressiveness of the implementation of the computational contract system, the programmer can intercept more or fewer execution points of the internal workings of the contracted entity.

A computational contract C_c over a function is denoted as follows, $C_d \xrightarrow{C_c} C_r$. The computational contract C_c is verified in the dynamic extent of the execution of each function application. In order to make this more concrete reconsider the `sqrt` function shown in Section 2. This function should not display anything to the user. With computational contracts this behaviour can be enforced by specifying a contract over the `sqrt` function as shown in Figure 4. The contract defined over the `sqrt` function again specifies that the argument has to be a positive number and that the return value has to be positive. In addition the computational contract, denoted by $!call(display)$, specifies that during the execution of the `sqrt` function, `display` should not be applied. In case `display` is called during the function application of the contracted `sqrt` function, the `sqrt` function is blamed.

Prohibition of certain function invocations within the dynamic extent of a function application is only one type of computational contract. In the next section we show how to define and use more advanced computational contracts in Racket.

4. Applications of Computational Contracts

Current pre/post contracts consider the function over which they are defined as a black box. This reduces the expressiveness of pre/post contracts to the verification of the argument and return-value(s) of the contracted function. However, many functional and non-functional requirements can only be expressed by verifying well-defined execution points *during* the execution of the contracted function. One example of such a requirement, to prevent a certain function to be applied, was described in Section 3. In this section we show how to specify and use the computational contract that verifies this requirement. Subsequently we show how to define and use a computational contract that verifies mandatory function calls. Next we show computational contracts that go beyond single function applications. We show how to express computational contracts that verify that the order of function applications within a contracted function follows a usage protocol. Finally we show how to implement computational contracts that can verify complex usage protocols, such as output volume or memory consumption.

4.1 Prohibit Contracts

In order to specify a computational contract that prohibits a function to be applied during the execution of a contracted function, the developer only has to specify which function call is disallowed. With computational contracts prohibiting a function call is done by generating a contract with the function `prohibit/c`. For example, `(prohibit/c (call f))` prohibits the function `f` to be applied within the dynamic extent of the contracted function. Note that in this definition `(call f)` denotes applications of the function that the variable `f` refers to.

```
1 (provide/contract
2  [sqrt (prohibit/c  (call display))])
```

Figure 5. Using a prohibit contract to prevent the `sqrt` to apply display.

```
1 (provide/contract
2  [map-pos
3   ((and/c
4     (prohibit/c  (call display))
5     (greater-than-zero? . → . greater-than-zero?))
6    (listof greater-than-zero?)
7    . → .
8    (listof greater-than-zero?))])
```

Figure 6. Using the prohibit contract to prevent the argument to the `map-pos` function to apply display.

Contracts are most effective at module boundaries [12]. The Racket module system supports exporting functions and contracting them at once by using `provide/contract`. This construct expects one or more lists where the first element is the function to export and the second element the contract that is defined over the exported function. In our examples all contracted functions are defined in a file called `"defs.rkt"` and exported with a contract. These functions are then applied from a different module in a file called `"uses.rkt"`. Computational contracts are tightly integrated with the Racket contract system. For example the `prohibit/c` function can be used to contract exported functions in Racket as shown in Figure 5. In this example the function `sqrt` is exported with the prohibit contract that ensures that the function `display` is not applied. The prohibit contract assigns blame to the function `sqrt` whenever the function `display` is applied in the dynamic extent of the `sqrt` function.

The same `prohibit/c` contract constructor can also be used to contract functional arguments. Let us revisit the example of the `map-pos` function from Section 2.2 and add a prohibit contract over the supplied function as shown in Figure 6. Combining the existing functional contract (`greater-than-zero? . → . greater-than-zero?`) with the computational contract is done by making use of the standard `and/c` function of Racket. This function expects two contracts and produces a new contract that verifies both contracts.

When using the contracted `map-pos` function correctly it behaves like any other function. A transcript that shows an example where the function `map-pos` is applied to the increment function and the list `'(1 2 3 4)` is shown below. The result of this function is as expected the list `'(2 3 4 5)`.

```
1 Welcome to DrRacket, version 5.0.1 [3m].
2 Language: racket [custom]; memory limit: 128 MB.
3 > (map-pos add1 '(1 2 3 4))
4 '(2 3 4 5)
```

When the `map-pos` function is applied to a function that violates the computational contract, i.e. applies `display`, blame is assigned to the caller of the `map-pos` function. A transcript of this interaction is shown below. Instead of the list `'(2 3 4 5)` an error message is presented. This error message shows that the violation was caused by the file `uses.rkt`. As the transcript was taken from the interaction window of the `uses.rkt` file, it can be easily deduced

that the blame is assigned to the call made from the prompt[4]. It is important to note that blame is assigned when the argument function is applied in the body of the map-pos. When the contracted function is applied it is in general not possible to determine that the function passed as an argument will behave according to the contract. This is also the main reason why blame assignment is needed in the context of higher-order functions. Because violations can not be detected when the contracted function is applied, verifying the contract is postponed. When a violation is detected during the execution of a contracted function, it is not always clear whether this is the fault of the caller or of the callee. Blame assignment solves this problem.

```
1  > (map-pos (lambda (x)
2              (display x) (+ x 1)) '(1 2 3 4))
3          ...
4   (file uses.rkt)
5   broke the contract
6   (->
7    (and/c
8     (-> greater-than-zero? greater-than-zero?)
9     prohibit-call-display)
10   (listof greater-than-zero?)
11   (listof greater-than-zero?))
12  on map-pos; computational contract violation ...
```

In DrRacket, the same error output is shown when evaluating the `uses.rkt` file that contains the same code as typed into the prompt. To pinpoint which line in the `uses.rkt` file caused the violation of the computational contract the stack trace can be used. An excerpt of this stack trace is shown below. It reveals that the origin of the violation was in the `uses.rkt` file on line 36, character zero. Also shown in the stack trace is that this line contains the call to the `map-pos` function.

```
1 .../uses.rkt: 36:0
2   (map-pos (lambda (x)
3   (display x) (+ x 1)) '(1 2 3 4))
```

Note that the function `display` was never applied. The blame assignment of the computational contract stops the current evaluation and prevents the function `display` to be applied completely.

4.2 Promise Contracts

The dual of prohibiting an action is to *promise* to perform an action. A promise contract[5] verifies that a certain promise is kept during the dynamic extent of the contracted function. For example, a function `g` can promise to apply another function `f`. Verifying a promise contract is more subtle than verifying a prohibit contract as blame can only be assigned *after* the contracted function returns. Defining a promise contract with computational contracts is as simple as defining a prohibit contract. Promise contracts are created with the function `promise/c`. When applied to `(call f)` it returns a promise contract that verifies that the function `f` is applied within the dynamic extent of a contracted function.

To exemplify the use of a promise contract consider the function `display-average` shown in Figure 7. This function takes a list of numbers and displays the average of the list. It is exported with a promise contract that assigns blame when the promise of applying the function `display` is not held.

This function `display-average` correctly computes the average of the list passed as an argument. Unfortunately our contracted function does not apply the function `display`. Therefore applying

```
1 (define (display-average a-list)
2   (/ (foldl (lambda (x acc) (+ x acc)) 0 a-list)
3      (length a-list)))
4
5 (provide/contract
6  [display-average
7   (and/c
8    (promise/c (call display))
9    (-> (listof integer?) any/c)) ])
```

Figure 7. Exporting the function `display-average` with a promise contract.

```
1 (define open-close-protocol
2   (protocol init
3             [init : ((call open-input-file)
4                      -> more)]
5             [more : ((call close-input-port)
6                      -> end)]
7             [end : accept]))
```

Figure 8. Defining a protocol for opening and closing files.

the list `'(10 20)` to the function `display-average` leads to a violation of the promise contract. A transcript of this example is shown below. As highlighted in the transcript the function display-average violates the `promise-call-display` contract.

```
1 > (display-average '(10 20))
2   (file .../defs.rkt)
3  broke the contract
4   (and/c
5    promise-call-display
6    (-> (listof integer?) any/c))
7  on display-average;
8  computational contract violation ....
```

4.3 Usage Protocols

Computational contracts can also be used to verify that a usage protocol is respected. Until now we have only considered computational contracts where the application of a single function is verified, i.e. constructed with `(call f)`. In this section we show computational contracts where the programmer describes a partial specification of the path of applications a certain function should or should not follow. In our implementation this path is expressed by a finite state machine[6]. As shown in Figure 8, this finite state machine describes which functions can be applied successively.

The protocol specifies that the end state can be reached after exactly one application of the function `open-input-file` followed by one application of the function `close-input-port`. Functions that are not specified in the protocol can always be applied. For example the application sequence `open-input-file`, `display`, `close-input-port` leads to the end state.

When an application sequence does not follow the protocol blame is assigned. For example, the application sequence `open-input-file`, `close-input-port`, `open-inputfile` is not allowed as the end state does not allow any applications to `open-input-port`.

Once a protocol is defined it can be used to create a computational contract with the `promise/c` function. When this function is applied to a protocol it returns a new computational contract. The resulting computational contract assigns blame to the contracted function when the function applications in the dynamic extent of

[4] The name `prohibit-call-display` in the error message is derived by the contract system.

[5] Not to be confused with promises as defined by Friedman et al. [14]

[6] We use a macro in the line of [28] to define the finite state machine.

```
1 (define (read-char-from-file filename)
2   (let ((port (open-input-file filename)))
3     (read-char port)))
4
5 (provide/contract
6  [read-char-from-file
7   (and/c (promise/c open-close-protocol)
8          (-> string? char?))])
```

Figure 9. Defining a protocol contract over the `read-char-from-file` function.

```
1 (define create-window-protocol
2   (protocol init
3             [init : ((call create-window)
4                      -> more)]
5             [more : ((call create-window)
6                      -> end)]
7             [end : accept]))
```

Figure 10. Open windows protocol definition.

the contracted function *do not obey* the usage protocol. This happens when functions are applied in the wrong order or when the finite state machine is not in the end state when the contracted function returns.

To exemplify the use of the `open-close-protocol` consider the `read-char-from-file` function shown in Figure 9. This function opens a file and reads one character from this file. The `read-char-from-file` function is exported with a contract that promises that the `open-close-protocol` is followed. Further it is specified that the argument of the function should be a `string` and the return value a `char`.

Applying the exported `read-char-from-file` function results in an error message as shown below. In the error message we see that blame is assigned to the file `defs.rkt`. It is also specified that the violation was a `promise-protocol/c` contract violation. Further the error indicates that the finite state machine was not in the end state and that the last application was the `open-input-file` function. From this information it is easy for the programmer to determine the root of the problem and adjust the `read-char-from-file` function so that it closes the file after reading.

```
1   (file .../defs.rkt)
2  broke the contract
3   (and/c promise-protocol/c (-> string? any/c))
4  on read-char-from-file; promise-protocol/c
5  violated, not in end state, last transition
6  after 'open-input-file
```

A computational contract that prohibits following a protocol can be created with the function `prohibit/c`. Such a computational contract assigns blame to the function over which it is defined when the function applications in the dynamic extent of the contracted function obey the usage protocol. This happens when all function applications are applied in such an order that the finite state machine reaches the end state. To show a use of a prohibit-protocol contract consider the protocol shown in figure 10. This protocol reaches the end-state after exactly two applications of the `create-window` function. When applying this protocol to the `prohibit/c` function it returns a contract that prohibits a function to create more than one window. From the moment the contracted function creates two windows it violates the prohibit contract because this application sequence leads to the end state of the finite state machine.

```
1 (define (make-and-show-window filename)
2    ...
3    (create-window ...)
4    (create-window ...)
5    ...
6    )
7
8 (provide/contract
9  [make-and-show-window
10   (and/c (prohibit/c create-window-protocol)
11          (-> string? any/c))])
```

Figure 11. Defining a prohibit protocol contract over the `read-char-from-file` function.

The function `make-and-show-window` shown in Figure 11, expects a filename and shows the content to the user. In order to make sure that this function does not create more than one window it is exported with a prohibit contract: `(prohibit/c create-window-protocol)`. Using the function `make-and-show-window` leads to a violation of the prohibit contract as shown in below.

```
1   (file .../defs.rkt)
2  broke the contract
3   (and/c prohibit-protocol/c (-> string? any/c))
4  on make-and-show-window ; prohibit-protocol/c
5  violated, reached end state.
```

The use of protocols allows the programmer to express certain quality of service contracts. For example, in order to avoid service abuse the programmer can define a protocol that states that a function must be called at least twice and at most five times.

4.4 Usage Protocols with Context

The protocols shown in the previous section are stateless. However, the verification of many concerns depends on a *context* or state, which has to be updated during the execution of the contracted function. For example verifying the memory consumption of a function depends on the amount of memory that is allocated while executing the function. With a `state-protocol` the programmer can define protocols that pass a certain context value when moving from state to state. Updating this context value often depends on the arguments of the functions that are being monitored by the protocol. For example when monitoring memory consumption, the argument of applying the `malloc` function determines with how much the context value should be increased. One application of `state-protocol` contracts is the verification of quality of service contracts *during* the execution of a function.

In order to make this more concrete, let us consider a computational contract that monitors the sound volume settings and verifies that the sound volume is not increased above a certain threshold S_t. In order to clearly illustrate the functionality of this contract consider Figure 12. In this graph the sound volume setting over time is depicted. As can be seen at a certain moment the function `f` is applied. During the execution of function `f` the sound volume increases (above S_t) however before the function returns, the sound volume decreases again below the threshold. Because a computational contract can monitor the sound volume settings *during* the execution of the function `f`, it can assign blame at the moment that the sound volume is higher than S_t. Pre/post contracts are not able to monitor such concerns. The only effect that they can measure is the difference in sound volume between the start and the end of the function application as discussed in Section 2.3.

Implementing the sound volume contract requires the interception of the volume-up and volume-down functions while keeping track of the increased sound volume. The protocol that ver-

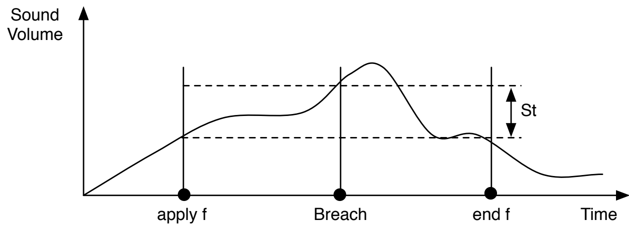

Figure 12. Sound volume contract violation.

```
1 (define sound-protocol
2   (state-protocol
3    (check 0)
4    [(check volume (< volume 100)) :
5     (((call volume-up)  up) ->
6      (check (+ volume up)))
7     (((call volume-down) down) ->
8      (check (- volume down)))]))
```

Figure 13. Sound volume contract definition.

```
1 (define (increase-sound)
2         (volume-up! 20)
3         ...
4         (volume-up! 200)
5         ...
6         (volume-down! 150))
7
8 (provide/contract
9  [increase-sound  (promise/c sound-protocol)])
```

Figure 14. Exporting the function `increase-sound` with the sound protocol contract.

ifies this sound volume contract is shown in Figure 13. Unlike previous protocols, the sound volume protocol is defined using a `state-protocol`, because it needs to keep track of a context value; in this case, the increased sound volume. The syntax of a `state-protocol` is similar to the syntax of a normal protocol. On line 3 of Figure 13 the sound protocol defines the initialisation state. As a `state-protocol` keeps track of a certain context value the initial state needs to be applied to an initial context value. In this case, the start state is the `check` state and the initial context value is the number 0. The start state, defined on line 4, binds the context value to the variable `volume` and is guarded by the predicate `(< volume 100)`. When a transition to the check state passes a context value higher than hundred blame is assigned. The check state defines two transitions, one for when the function `volume-up` is applied and one for when the function `volume-down` is applied. When the `volume-up` function is applied to the value `up` the new state is again the `check` state but during the transition the context value `volume` is increased with `up` (line 6). Similarly when the `volume-down` function is applied the context value is decreased.

In Figure 14 an example use of the `sound-protocol` contract is shown. In this code the function `increase-sound` is exported with the `sound-protocol` contract. Increasing the volume with two-hundred results in a violation of the contract. Therefore, when applying this function the programmer is presented with an error message as shown below. This error message shows that the function `increase-sound` violated the computational contract when moving to the state `check`. From this error message the developer can derive that during the execution of the function `increase-sound` the volume was increased over the value `100`.

```
1  (define (flat pred?)
2    (lambda (pos neg)
3      (lambda (val)
4        (if (pred? val) val (blame pos)))))
5
6  (define (ho dom rng)
7    (lambda (pos neg)
8      (lambda (f)
9        (if (procedure? f)
10           (lambda (argument)
11             ((rng pos neg)
12              (f ((dom neg pos) argument))))
13           (blame pos)))))
14
15 (define (guard ctc val pos neg)
16   ((ctc pos neg) val))
```

Figure 15. Higher-Order pre/post contract constructors.

```
1   (file .../defs.rkt)
2  broke the contract
3  promise-protocol/c
4  on increase-sound;
5  protocol computational contract violation
6  when moving to state check.
7  #<procedure:...defs.rkt:138:53>
```

5. Contract Verification and Blame Assignment

In this section we show the inner workings of the computational contract system by presenting a minimal implementation. As computational contracts are an extension to higher-order pre/post contracts, we first start with a detailed explanation of the higher-order pre/post contract system. Readers who are already familiar with the implementation of higher-order pre/post contract systems are still encouraged to skim through this section in order to get familiar with our notation. After the explanation of higher-order pre/post contracts we show how to extend them in order to support computational contracts.

5.1 Flat and Higher-Order Pre/Post Contracts

The higher-order pre/post contract system implementation consists of three functions, `flat`, `ho` and `guard`[7]. These functions are shown in Figure 15. The function `flat` consumes a predicate and creates a contract that verifies this predicate. Flat contracts are used to contract simple values. The function `ho` creates a functional contract given a contract for the domain and a contract for the range. Finally the function `guard` creates contracted values. The first argument is a contract, created with either `flat` or `ho`. The second argument is the value over which the contract is defined. Finally the last two arguments are *blame labels*, *i.e.* textual representations of the supplier and consumer of the contracted value. These labels are passed to the contract in order to assign blame in case of a violation.

In order to get a better understanding of the contract verification mechanism in combination with blame assignment consider the example shown in Figure 16. In this example two functions `fix10?` and `neg` are defined. The function `fix10?` takes a function and verifies that `10` is a fixed point of the given function. The function `neg` returns the negative value of integers passed as argument. The example also shows the creation of two flat contracts, one with the predicate `positive?` and one with the predicate `boolean?`. Finally a new contracted version of `fix10?`, `cf`, is defined by making use of the function `guard`, `ho`, and the newly created

[7] We adopt the implementation as presented in [10].

```
1 (define (fix10? f)  (= (f 10) 10))
2 (define (neg x) (display x) (* -1 x))
3
4 (define int? (flat positive?))
5 (define bool? (flat boolean?))
6
7
8 (define cf (guard (ho  (ho int? int?) bool?)
9                   fix10?
10                  "fix10?"
11                  "prompt"))
```

Figure 16. An example higher-order contract definition with the minimal implementation.

`flat` contracts. This functional contract states that the argument of `fix10?` is a function that takes a positive value and returns a positive value. The contract also states that the function `fix10?` returns a boolean value.

The function `guard` applies the higher-order pre/post contact by passing the higher-order pre/post contract the blame labels and the function `fix10?`. The result is a contract verification function that behaves almost exactly the same as the function `fix10?` with the difference that it verifies the domain (`int?` $\rightarrow$ `int?`) and range (`boolean?`) contract.

In our notation, wrapping a function into a contracted function is represented by a box. Wrapping of the function `fix10?` into a contracted function is represented as follows:

$$\boxed{l_1, l_2|\ \texttt{(int?}\ \rightarrow\ \texttt{int?)}\ \rightarrow\ \texttt{bool?}\ |\ \texttt{fix10?}}$$

where l_1 and l_2 are the blame labels textually representing the supplier and the consumer of the contracted value respectively.

In order to show how blame assignment of contracted functions works consider the following example of applying the contracted version of `fix10?` to the function `neg`:

$$(\boxed{l_1, l_2|\ \texttt{(int?}\ \rightarrow\ \texttt{int?)}\ \rightarrow\ \texttt{bool?}\ |\ \texttt{fix10?}}\ \texttt{neg})$$

Here l_1 is the blame label of `fix10?`, and l_2 is that of the prompt.

When the contracted function `fix10?` is applied it verifies the domain contract, namely that the supplied function `neg` transforms positive values into positive values. Because this can not be checked immediately when `fix10?` is applied, the `neg` function is wrapped into a new contracted function. Note that for this new contracted function, the blame labels are swapped. An intuitive explanation for this blame label swapping is that within the function body of the function `fix10?` the prompt is the supplier of the function `neg` and the contracted function is the client. After wrapping the function `neg` into a contracted function, the function `fix10?` is applied as shown below:

$$\texttt{(fix10?}\ \boxed{l_2, l_1|\ \texttt{int?}\ \rightarrow\ \texttt{int?}\ |\ \texttt{neg}})$$

As shown in Figure 16, within the body of `fix10?` the function `neg` is applied to the value 10. Since the argument is now a contracted function, its domain contract is verified first:

$$(\boxed{l_2, l_1|\ \texttt{int?}\ \rightarrow\ \texttt{int?}\ |\ \texttt{neg}}\ \texttt{10})$$

In this case the argument is a simple value (10) and the domain contract `int?` verifies that this value is indeed a positive integer value. As the supplied value, 10 in the example, passed the domain contract, the function `neg` is applied and the return value (-10) is verified by the range contract. This value does not pass the flat contract `int?`. Therefore blame is assigned to the prompt, as the prompt was the supplier of the function `neg`, which violated the contract.

```
1 (define (cc pc domain range)
2   (lambda (pos neg)
3     (lambda (f)
4       (if (procedure? f)
5           (lambda (argument)
6             (let ((verified
7                    ((domain neg pos)  argument))
8                   (result '()))
9               (deploy-fluid
10               (aspect pc (make-blame-advice pos))
11               (set! result (f verified)))
12              ((range pos neg) result)))
13          (blame pos)))))
```

Figure 17. Computational contract constructor.

5.2 Computational Contracts Implementation

An important aspect of computational contracts is how to define and intercept erroneous behaviour during the execution of a computation. Therefore we have focused our effort to make this interception mechanism as separated form the contracted entity as possible. The characteristics of computational contracts have led us to the use of aspect-oriented programming [27].

Aspect-oriented programming introduces a technique that allows the programmer to modularise concerns that are normally scattered throughout the code. As these concerns "*cut through*" the code they are referred to as crosscutting concerns. Aspect-oriented programming allows the specification of crosscutting expressions in a modular way so that they are no longer scattered through the code but localised in one place of the code. These modular crosscutting expressions are defined by the specification of additional behaviour called *advice* on particular points in the programs execution called *join points*. The developer specifies on which set of join points the advice has to be executed by specifying a *pointcut*.

In the implementation of computational contracts, the concepts of pointcut and advice are used to specify at which point the contract has to be verified. By using the techniques established in aspect-oriented programming this can be done without having to make any *intrusive* changes to the contracted entity.

In order to achieve the particular behaviour of only enforcing the computational contract in the dynamic extent of the function over which it is defined, dynamically-scoped aspects [8] are used. Instead of being statically and globally defined at the start of the program, aspects can be deployed only over certain parts of the execution of the program.

In Figure 17, the computational contract constructor function `cc` is shown. Note that this function is used to create prohibit contracts. The first argument of this function is a pointcut that specifies at which join points the contract is checked. The two last arguments are the domain contract and the range contract. A computational contract verifies its contracted function very similarly to how higher-order pre/post contracts are verified. The difference is that after verifying the domain contract, a dynamic aspect is deployed. The aspect intercepts violations of the computational contract (with `deploy-fluid`[8]). Blame is assigned to the contracted function when a matching join point is encountered in the dynamic extent of applying the contracted function. The internal implemen-

[8] In practice, we use LAScheme as a concrete aspect language for Racket [42]. LAScheme is directly based on AspectScheme [8], but integrates a number of improvements, such as execution levels [43].

```
1 (define cf
2   (guard
3    (ho (cc (call display) int? int?)
4        int?)
5    fix10?
6    "fix10?"
7    "Prompt"))
```

Figure 18. Minimal computational contracts in use.

tation of LAScheme makes use of preserved thread fluids (parameters) [16], in order to make sure that continuation capturing and subsequent invocation of these continuations are handled correctly. When there is no violation of the computational contract during the execution of the contracted function the computational contract behaves exactly like a higher-order pre/post contract.

5.3 Verification and Blame Assignment in Computational Contracts Step by Step Example

In order to show the impact of computational contracts on the execution of a contracted function let us revisit the example of the `fix10?`, shown in Figure 16. However now the contract defined over the argument function `fix10?` prohibits calls to `display` during its execution as shown in Figure 18. In order to show how blame assignment of computational contracts works, consider the following example of applying the contracted function `fix10?` to the function `neg`:

$$(\boxed{l_1, l_2|\ \texttt{(int?} \xrightarrow{!call(display)} \texttt{int?)} \rightarrow \texttt{bool?}\ |\ \texttt{fix10?}}\ \texttt{neg})$$

When the contracted function `fix10?` is applied it verifies the contract, namely that the supplied function `neg` does not apply the function `display`. As explained before, this can not be verified when the contracted function is applied. Therefore, the argument is wrapped into a new contracted function. The contracted function `fix10?` is applied to the contracted `neg` as shown below.

$$(\texttt{fix10?}\ \boxed{l_2, l_1|\ \texttt{int?} \xrightarrow{!call(display)} \texttt{int?}\ |\ \texttt{neg}})$$

Within the body of the `fix10?` function this contract-verification function is applied to the value 10. This results in verifying the domain contract `int?`, as in the previous example the value 10 passes this contract. Finally the computational contract is applied.

$$(\boxed{l_2, l_1|\ \texttt{int?} \xrightarrow{!call(display)} \texttt{int?}\ |\ \texttt{neg}}\ \texttt{10})$$

Applying the computational contract corresponds to deploying a dynamic aspect. The active pointcut is shown in superscript ($!call(display)$) and the responsible to blame in subscript (l_2).:

$$\texttt{(neg 10)}_{l_2}^{!call(display)}$$

In this example the function `neg` violates the computational contract by applying the function `display`. This results in the pointcut to match and the blame advice to be executed. The blame in this case is assigned to the prompt. This is correct because the prompt supplied the function `neg` which does not obey the contract.

6. Discussion

We now discuss some subtleties that arise in the interaction between computational contracts and existing contracts.

```
1 (define (process f) ... (f 4) ...)
2 (define (remove path)  ... )
3
4 (provide/contract
5  [process (prohibit/c (call remove))]
6  [remove  (-> string? boolean?)])
```

Figure 19. Example where a computational contract and a pre/post contract can be active over the *same* function.

6.1 Blame Precedence

It is possible that a function is subject to verification by both a computational contract and a pre/post contract at the same time. In that case, the question of which has precedence appears.

To clarify this, consider the code example show in Figure 19. There are two functions defined, `process` and `remove`. For the discussion it is sufficient to know that `process` applies the function `f` given as argument and that the function `remove` destructively deletes a file from the harddisk. Both functions are exported with a contract. `process` has a computational contract that prohibits the function `remove` to be applied. The function `remove` has a higher-order contract that states that the arguments of the function `remove` should be a `string` and that the return value should be a `boolean`.

Let us consider that the exported function `process` is applied to the exported function `remove` from the prompt. Remember that exporting a function with a contract creates a new function that acts and behaves almost exactly like the original function, with that difference that the contract is verified. Following the notation introduced in Section 5, the function `process` in that application is represented as follows (l_1 is the blame label for `process`, and l_2 for the prompt):

$$\boxed{l_1, l_2|\ \texttt{(any/c} \xrightarrow{!call(remove)} \texttt{any/c)}\ |\ \texttt{process}}$$

Similarly the function `remove` is represented as follows (l_3 is the blame label for `remove`):

$$\boxed{l_3, l_2|\ \texttt{(string?} \rightarrow \texttt{boolean?)}\ |\ \texttt{remove}}$$

Applying the exported function `process` to the exported function `remove` results in the computational contract to be verified. After deploying the computational contract the function body of `process` is executed (line 1). In the body, the function argument `f` is applied to the number 4. Graphically, we have:

$$(\boxed{l_3, l_2|\ \texttt{(string?} \rightarrow \texttt{boolean?)}\ |\ \texttt{remove}}\ \texttt{4})_{l_1}^{!call(remove)}$$

Note that the function `remove` is still contracted by the pre/post contract `(string?` $\rightarrow$ `boolean?)`. At the same time a computational contract that prohibits applications of the function `remove` is also active (indicated in superscript). Evaluating either contract leads to a violation. The question is: which contract has precedence?

When precedence is given to the computational contract, blame is assigned to the function `process` and a computational contract violation is presented to the developer. For the developer it will be clear that some piece of code attempted to remove parts of his harddisk while the contract clearly prohibits this. When precedence is given to the pre/post contract, blame is assigned to the module where the function `remove` was applied from. In this case the developer is presented with a precondition violation as the function `remove` is applied to a number instead of a string. A developer presented with this error message could be tempted to correct this error. Of course such an attempt would be futile as applications of

the function `remove` are prohibited by the computational contract anyway.

In our implementation, by default, pointcuts like `(call f)` select applications of a function `f`, whether or not it is contracted; under the hood, it relies on Racket's `equals?` function, which makes equality oblivious to contracts. This means that in the previous example, the programmer would get a computational contract violation. We also provide a `call-eq` pointcut designator, which relies on the low-level equality function `eq?`. In that case, the computational contract is not applied before the pre/post contract; hence the programmer gets a precondition violation.

Finally, note that a function can also be subject to verification by multiple computational contracts at the same time. In our implementation we always give precedence to the computational contract that has been deployed last. The reasoning behind this decision is that in case of multiple applicable violations the computational contract which is the "closest" to the violation will be presented to the programmer. We have not yet encountered any scenario where changing the precedence of the computational contracts makes sense. However, it would not be hard to support custom precedence declarations.

6.2 Who will Guard the Guards?

An important aspect of contract systems is whether they assume that the contracts themselves are trustworthy or not. Dependent contract as described in Findler and Felleisen's original paper [12] do not enforce the domain contract defined over the arguments during the evaluation of the postcondition. Dependent contracts thus fall into the category of contracts where a contract is assumed to be always correct. This was criticised by Blume and McAllester [3]. They extended the work on depended contracts so that the domain contract is enforced both in the precondition *and* in the postcondition of the dependent contract. Blume and McAllester's contract system is dubbed *picky* while Findler and Felleisen's original dependent contracts are called *lax*. While *picky* contracts capture more violations they do not assign blame to the contract. Recently Dimoulas et al. [4] have further extended the *picky* blame assignment. This system dubbed *indy*, treats the contract as an *independent* party and in case that the postcondition violates the domain contract, blame is assigned to the contract.

A very similar phenomena is observed when working with computational contracts. During the verification of a computational contract the pre/post contract might violate the computational contract. An example of this is shown below.

$$(\boxed{l_1, l_2|\ \texttt{(printArgument} \rightarrow \texttt{int)}\ |\ \texttt{foo}}\ \texttt{1})^{!call(display)}$$

The precondition, `printArgument` simply allows any argument to pass but also `displays` the argument. The computational contract however disallows any application of the `display` function. As of now, our implementation provides lax computational contracts, as they would allow the above behaviour. Adapting the notion of indy contracts to computational contracts is future work.

7. Related Work

Design by contract has been explored in a plethora of programming languages and programming abstractions. Related work therefore spans a number of different areas which we have categorised into four topics. The first topic is about contract frameworks with direct support for design by contract in a pre/post contract fashion. The second topic is about QoS contracts, which allow more expressive contracts to be specified. The third topic is about grey box verification techniques, which allow contracts to reason about the execution of contracted entities. Finally we highlight how aspect-oriented programming has been applied for contract verification.

7.1 Programming Support for Design by Contract

Contract frameworks have been around for a very long time but the first appearance of pre and post-conditions can be found in the work of Tony Hoare [23]. These ideas were first adopted and implemented in the programming language CLU by Barbara Liskov [32]. Afterwards they were adopted in a wide range of programming languages but were mostly influenced by the programming language Eiffel [33], which also introduced the term Design by Contract (DbC). Ada supports DbC via pre-compiler statements for preconditions and postconditions.

Many languages provide something similar to contracts with special predicates called assertions. The difference between assertions and contracts is that contracts apply restrictions on the boundaries of components whereas assertions can show up in arbitrary positions in the code. Depending on the system these assertions are verified at runtime or statically. When a violation of the assertion is detected at runtime, an error is thrown, and it is up to the developer to assign blame. Support for assertions is included in the C programming language [26], and consequently C++ [40]. The .NET Framework includes support for assertions in the debug class since version 1.1. The first drafts of the Java language specification called Oak already had support for assertions but they never made it into the final specification. Only in version Java 1.4 [17] assertions were added but a real DbC framework is still not in the main language.

The large number of external frameworks which provide additional support for DbC like JML [30], ContractJava [11], Handshake [7], KJC [29], Jcontract [25] only confirms the high demand for design by contract. Programming languages running on the JVM including Groovy & Clojure also have implemented DbC. DbC is supported by most popular scripting languages including JavaScript, Ruby, Python,Tcl.

Functional programming languages have adopted support for DbC, for example in OCaml with pre- and post conditions and in Common Lisp via the meta object protocol. Finally there is a whole line of work [3, 18, 19, 38] spawned from the higher-order contracts as highlighted in section 5.1. While the wide range of programming languages presented here clearly shows the need for DbC, all these contract frameworks are basically pre/post contract systems and do not allow the programmer to specify any constraints over the actual computation.

7.2 QoS Contracts

In the component-based middleware community, contracts have been incorporated in order to compose and adapt applications in order to meet a certain quality of service. QoS frameworks such as QuO [48], QML [15] and 2KQ+ [31], provide abstractions in order to enforce a contract over the *bindings* between a client and a server component. Depending on the implementation, a QoS contract verifies quality constraints such as latency, duration of a computation, throughput etc. The contracts in such systems mainly describe how the component should adapt itself depending on these quality constraints. The typical example of such frameworks is to switch between compressing an image when the connection is slow or sending the uncompressed image when the connection is fast. The monitoring of such QoS constraints can happen on the application of certain functionalities of the component or during their execution. Several QoS contracts have made use of aspect technology in order to adapt the application according to the context [15, 20]. However the focus of these contract systems is to adapt the application such that a certain QoS can be offered. They do not verify the computation of the components themselves in order to assign blame in case of a violation.

7.3 Grey Box Verification Techniques

There is a group of research frameworks that focuses on grey box verification techniques. Similar to our work, these verification mechanisms allow the programmer to define more expressive verification statements than simple pre/post conditions. Helm et al. [22] and Holland [24] were among the first to use such advanced mechanisms. Their approach uses model programs in order to describe contractual specifications, but they do not present a method for automatic conformance monitoring.

Shaner et. al. have extended JML for higher-order methods (HOM) [37]. They define a higher-order method as any method whose behaviour critically depends on one or more mandatory calls. This approach focuses on static verification and does not support higher-order contracts, i.e contracts over the argument values can not be specified. A related approach by Soundarajan and Tyler [44] allows trace-based specifications but suffers from the same limitations as Shaner's system.

MaC [5] is a runtime verification system where program executions points, such as the application of a function, are reified as events. Over these events the programer can write rules in order to verify the program execution. While it is likely that the expressive power of the MaC system allows computational contracts to be defined it has not been designed for higher-order functions. Blame assignment is also not considered.

A very related approach by Fischer [13] introduces trace-based assertions. These are similar in nature to the contracts for monitoring the sound volume shown in Section 4.4. However trace-based assertions do not support functional contracts to be defined over the argument values of a function.

A remarkable contract system that goes beyond pre- and post-conditions was recently proposed by Heidegger et al. [21]. They propose access permission contracts, which allow programmers to annotate methods with a set of read and write access paths. During the execution of a contracted function the dynamic extent of the contracted function can only read and write to those variable in their access paths. Access permission contracts are a particular instantiation of computational contracts and we plan to implement them in our framework as future work.

Finally, Higher Order Temporal (HOT) Contracts [6] extend prior higher-order contract systems to also express and enforce temporal properties between modules. In their formalisation, module behaviour is modelled as a trace of events such as function calls and returns, which does not include internal module calls. This has two major implications. First, the `prohibit/c` contract shown in Figure 5 and similar contracts are not expressible with a HOT contract because internal calls are not in the trace. Secondly, it is not only important to check that a client respects a given protocol, but also that a provider of the said protocol fulfils it. Because internal module calls are not in the trace, HOT contracts do not verify that the provider of their protocols fulfils it. This makes it possible to define a module that internally violates its own HOT contract but will never be blamed for it. With computational contracts all internal module applications are monitored.

7.4 Aspect-Oriented Programming and Contracts

Frameworks such as Barter [41], Jose [9] and Contract4J [46] have used aspect-oriented programming as an implementation technique in order to provide design by contract. Again these techniques do not support blame assignment in the context of higher-order programming languages. Several systems have added contracts for aspects [1, 39] where the focus lies on the definition of contracts over an aspect. For example, Pipa [47] extends JML [30] to support DbC for programs written with AspectJ. They do not focus on using aspects in order to verify certain properties of the computation. Finally, Contract-Based Verification for Aspect-Oriented Refactoring [45] uses aspects in order to specify contracts over refactorings.

8. Conclusion

Many aspects such as prohibiting or enforcing certain method invocations, access permission, time constraints, sending messages over the network, memory constraints etc. are well-defined properties of the computation of a certain function. However current higher-order contract systems do not provide a structured and expressive mechanism to verify these aspects. The core problem of current contract systems is that they treat a contracted entity as a black box. In this paper we introduced the notion of *computational contracts*. A computational contract is a contract over the execution of a contracted entity. In contrast to existing contracts, which treat a contracted entity as a black box, a computational contract can verify well-defined execution points *during* the execution of the contracted entity. With computational contracts the developer can define a functional contract that verifies a single event or a sequence of events during the execution of the contracted function. The developer can specify that certain events should or should not happen by making use of *promise* and *prohibit* computational contracts respectively. We have shown the inner workings of a minimal computational contract system by presenting the implementation in Racket. Computational contracts is the first contract model with blame assignment in a higher-order setting that provides a systematic way to perform grey box verification.

Acknowledgments

Christophe Scholliers is funded by a doctoral scholarship of the Institute for the Promotion through Science and Technology in Flanders (IWTVlaanderen). Éric Tanter is partially funded by FONDECYT Project 1110051.

References

[1] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 141–152, 2011.

[2] A. Beugnard, J.-M. Jezéquel, and N. Plouzeau. Contract aware components, 10 years after. *Electronic proceedings in theoretical computer science*, 37:1 – 11, October 2010.

[3] M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming.*, 16:375–414, July 2006.

[4] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 215–226, 2011.

[5] N. Dinesh, A. Joshi, I. Lee, and O. Sokolsky. Reasoning about conditions and exceptions to laws in regulatory conformance checking. In *Proceedings of the 9th international conference on Deontic Logic in Computer Science*, DEON '08, pages 110–124. Springer-Verlag, 2008.

[6] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *International Conference on Functional Programming*, pages 176–188, 2011.

[7] A. Duncan and U. Hoelzle. Adding contracts to Java with HandShake. Technical Report TRCS98-32, University of California at Santa Barbara, Santa Barbara, CA, USA, 1998.

[8] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, 63:207–239, December 2006.

[9] Y. A. Feldman, O. Barzilay, and S. Tyszberowicz. Jose: Aspects for design by contract. *IEEE International Conference on Software Engineering and Formal Methods*, pages 80–89, 2006.

[10] R. Findler and M. Blume. Contracts as pairs of projections. *Functional and Logic Programming*, 3945:226–241, 2006.

[11] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 1–15, 2001.

[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 48–59, 2002.

[13] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, Germany, Jan. 2000.

[14] D. Friedman and W. David. The impact of applicative programming on multiprocessing. *International Conference on Parallel Processing*, pages 263–272, 1976.

[15] S. Frølund and J. Koistinen. Quality of service aware distributed object systems. In *Proceedings of the 5th conference on USENIX Conference on Object-Oriented Technologies & Systems - Volume 5*, pages 69–83, 1999.

[16] M. Gasbichler, M. Sperber, and U. Tbingen. Processes vs. user-level threads in scsh. In *In Proceedings of the 3rd Workshop on Scheme and Functional Programming*, 2002.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice-Hall, 2005. ISBN 978-0321246783.

[18] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 353–364, 2010.

[19] A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric polymorphic contracts. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 29–40, 2007.

[20] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. Aspectix: A quality-aware, object-based middleware architecture. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 115–120, 2001.

[21] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts. *Unpublished*, 2011. URL `http://tiny.cc/hsxom`.

[22] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, OOPSLA/ECOOP '90, pages 169–180, 1990.

[23] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[24] I. M. Holland. Specifying reusable components using contracts. *European Conference on Object-Oriented Programming*, pages 287–308, 1992.

[25] M. Karaorman, U. Hölzle, and J. L. Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 175–196, 1999.

[26] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, pages 220–242, June 1997.

[28] S. Krishnamurthi. Educational pearl: Automata via macros. *Journal of Functional Programming*, 16(3):253–267, 2006.

[29] M. Lackner, A. Krall, and F. Puntigam. Supporting design by contract in Java. *Journal of Object Technology*, 1(3):57–76, 2002.

[30] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55:185–208, March 2005.

[31] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communications*, 17:1632–1650, 1999.

[32] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20:564–576, August 1977.

[33] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988. ISBN 0136290493.

[34] B. Meyer. *Eiffel : The Language*. Prentice Hall Object-Oriented Series, 1991. ISBN 0132479257.

[35] B. Meyer. Applying design by contract. *Computer*, 25:40–51, 1992.

[36] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[37] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 351–368, 2007.

[38] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 365–376, 2010.

[39] T. Skotiniotis and D. H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 196–197, 2004.

[40] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, special edition, 2000.

[41] V. Szathmary. Barter, beyond design by contract, 2002. URL `http://barter.sourceforge.net`.

[42] É. Tanter. LAScheme. http://pleiad.cl/research/lascheme, 2010.

[43] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010.

[44] B. Tyler and N. Soundarajan. Black-box testing of grey-box behavior. In *FATES'03*, pages 1–14, 2003.

[45] N. Ubayashi, J. Piao, S. Shinotsuka, and T. Tamai. Contract-based verification for aspect-oriented refactoring. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 180–189, 2008.

[46] D. Wampler. Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. *Fifth AOSD Workshop on ACP4IS*, 2006.

[47] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Fundamental Approaches to Software Engineering (FASE 2003)*, Warsaw, Poland, Apr. 2003.

[48] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems (TAPOS)*, 3(1):55–73, 1997.