Science of Computer Programming 98 (2015) 360–375

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Computational contracts

Christophe Scholliers [a,*,1], Éric Tanter [b,2], Wolfgang De Meuter [a]

[a] Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, Elsene, Belgium

[b] PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile

ARTICLE INFO

Article history:
Received 16 March 2012
Received in revised form 17 September 2013
Accepted 19 September 2013
Available online 31 October 2013

Keywords:
Higher-order contracts
Languages
Design
Reliability
Theory

ABSTRACT

Software contracts have proven to play an important role for the development of robust software. Contract systems are widely adopted in statically typed languages and are currently finding their entrance in dynamically-typed programming languages. Most research on higher-order contracts has adopted a black-box approach where only input and output are checked. These systems cannot check many interesting concerns about the behaviour of a function. Examples include prohibiting or ensuring that certain functions are called, checking access permissions, time or memory constraints, interaction protocols, etc. To address this need for behavioural runtime validation, while preserving support for higher-order programming, we introduce the notion of *computational contracts*. Computational contracts is a contract model with blame assignment in a higher-order setting that provides a systematic way to specify temporal contracts over objects and functions and their possibly higher-order arguments. We show various applications of computational contracts, and explain how to assign blame in case of a violation. Computational contracts have been integrated in both Scheme and AmbientTalk, a dynamically-typed object-oriented language built upon the principles of prototype-based programming.

1. Introduction

Design by contract (DbC) [22] is a software correctness methodology that is based on the principle of pre- and post-conditions to validate certain functionalities of a program. A contract specifies the obligations and the rights of both the implementor and the client who uses the contracted entity. Pre- and post-conditions existed a long time before the introduction of DbC, the novelty of DbC is that contracts are executable and defined by program code in the programming language itself. DbC was first popularised in the Eiffel [21] programming language and since then adopted in almost all mainstream languages (i.e. C++ [23], Java [19], C# [1]). Contracts for higher-order functions were introduced in Scheme by Findler and Felleisen [14]. Contracts have also been applied in multithreaded object-oriented systems to coordinate groups of objects [17].

In general the aim of contracts is to specify and validate well-defined properties of a system. Beugnard et al. [3] categorise contract systems in four levels: syntactic (type systems), behavioural contracts (pre/post conditions), temporal contracts (temporal ordering, time based synchronisation) and quality of service contracts (e.g. time and space guarantees). In this paper we focus on *runtime-validation of temporal contracts* for dynamically-typed object-oriented languages.

* Corresponding author.
E-mail addresses: cfscholl@vub.ac.be (C. Scholliers), etanter@dcc.uchile.cl (É. Tanter), wdmeuter@vub.ac.be (W. De Meuter).

1 Christophe Scholliers is funded by a doctoral scholarship of the Institute for the Promotion through Science and Technology in Flanders (IWTVlaanderen)..

2 Éric Tanter is partially funded by FONDECYT Project 1110051.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.09.005

```
def MathModule := {
 def positive := flat: {|x| x >= 0};
 def sqrt(x) { ... };
 def moduleInterface := object: {
  def sqrt := provide: sqrt withContract:  positive -> positive;
 };
};
```

Fig. 1. Simple MathModule providing a contracted `sqrt` function.

We have found that most contract systems do not provide a structured and expressive mechanism to check temporal aspects in a *higher-order setting*. Even though, many contract systems allow the programmer to define contracts at the class and/or interface level, making sure that an object received as an argument obeys a temporal contract is impossible in such a system if the object's class does not already enforces this behaviour.

A very simple example of the lack in expressiveness of current *higher-order* contract systems can be observed when implementing a temporal 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 validate 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 validating the postcondition. Even more importantly, checking this particular contract in the (post-condition) is too late i.e. the damage has already been done. Finally, current *higher-order* contract systems do not provide abstractions so that the programmer can easily detect that a file was read.

In this paper we present *computational contracts*, an extension to the higher-order contract systems defined by Findler and Felleisen [14]. Computational contracts tackle the problems of current higher-order contract systems by also allowing temporal contracts in dynamically-typed languages. Computational contracts allow the programmer to define and check temporal contracts over a function or an object and its higher-order arguments. We present various examples of computational contracts including mandatory function calls and protocol contracts. We describe an expressive model to specify and check higher-order computational contracts at runtime, including proper blame assignment. We implemented[3] computational contracts in AmbientTalk, a dynamically-typed object-oriented language built upon the principles of prototype-based programming. The work presented in this paper extends our initial work on computational contracts [25] for Scheme with support for object-oriented programming. In this paper we also give an operational semantics of computational contracts, which was not presented before.

We start our explanation of computational contracts by first giving a small overview of higher-order contracts in Section 2. Examples of computational contract applications are shown in Section 3. In Section 4, we illustrate the use of computational contracts for object protocols. Subsequently we give the operational semantics of computational contracts in Section 5. We discuss interactions between computational contracts and existing contracts in Section 7. Before concluding, we survey related work in Section 8.

2. AmbientTalk higher-order contracts in a nutshell

The AmbientTalk [6] contract system is based upon Findler and Felleisen's [14] 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 contract over the domain of the function and C_r is a contract defined over the range of the function. As the 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 a first-order function contract example, i.e., a contract 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 necessary to define contracts over (prototype) objects.

2.1. First-order function contracts

The prototypical example of contract frameworks is to define a contract over the `sqrt` function. The purpose of the contract is to ensure that the argument passed to the `sqrt` function is a positive number (pre-condition) and that the result of the `sqrt` function is also a positive number (post-condition). A possible specification of this contract in AmbientTalk is shown in Fig. 1.

Contracts in AmbientTalk are defined on the provided functionality of a module where they are the most effective [14]. The AmbientTalk module system supports exporting functions and defining a contract over them at once by using `provide:withContract:`. In the example the `MathModule` provides the function `sqrt` with a functional contract that is

[3] Available at http://soft.vub.ac.be/~cfscholl/index.php?page=at_contracts.

```
> import MathModule;
> sqrt("wrong");
>>(1:1:REPL)  sqrt("wrong") violated the contract,
expected positive given "wrong"
```

Fig. 2. The AmbientTalk interaction prompt: signalling a contract violation.

```
def map_pos(f, a) {
 a.map: f;
}
def moduleInterface := object: {
 def map_pos := provide: map_pos withContract:
 (positive -> positive) * arrayOf(positive) -> arrayOf(positive);
}
```

Fig. 3. Higher-order pre/post contract over the function `map_pos`.

```
>map_pos( {|x| x+1 }, [1,2,3])
>>[2, 3, 4]
>map_pos( {|x| "wrong" }, [1,2,3])
>> (1:1:REPL)  map_pos({ |x| "wrong"}, [1, 2, 3])  violated the contract,
expected positive given "wrong"
```

Fig. 4. The AmbientTalk interaction prompt: signalling a higher-order contract violation.

composed out of two flat contracts. These flat contracts are composed with the arrow operator (`->`). The `positive` contract is constructed by making use of the `flat:` contract constructor, which given a predicate function returns a flat contract. In AmbientTalk, literal closures are defined by curly braces and the formal arguments of the closure are written between bars (`|`). The flat contract over the argument of the `sqrt` function `positive` checks that the argument 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.

Modern higher-order contract systems have a mechanism that allows the responsible party to be pointed out in case a violation is detected. The process of figuring out who violated the contract is called *blame assignment* [14]. For first-order function contracts, 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). An example of using the contracted `sqrt` function from the `MathModule` is shown in Fig. 2. In this figure the AmbientTalk prompt is shown. First the math module is imported and then the `sqrt` function is applied to the string `"wrong"`. The contract system verifies the precondition and as expected assigns blame to the read–eval–print loop (1:1:REPL).

For a long time researchers agreed on assigning blame to the immediate context in which 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 (dynamic) 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 contracts can be used in order to define contracts over higher-order functions, i.e. functions that receive other functions as an argument or return functions. An example of a popular higher-order function is the map function, this function expects a function f and an array a. The map function creates a new array b where $b[i] = f(a[i])$. A higher-order contract that can be used to define a contract over a variation of the map function is shown in Fig. 3. This code excerpt specifies that the first argument of the `map_pos` function is a function, which expects a positive number and returns a positive number, the second argument of the `map_pos` function is an array of positive numbers. Finally the return value of the `map_pos` function must be an array of positive numbers.

As an example of the use of the `map_pos` function consider Fig. 4. In this example, the `map_pos` function is first applied correctly. Then the map function is applied again, however now the function passed as an argument does not produce positive integers. While the call to the map function is being evaluated the contract system detects this violation and assigns blame to the caller of the map function, in this case the `REPL`. This is in contrast to simple assertion-based systems, which cannot validate such contracts. In such a case an exception would be thrown when the values of the returned array are used in some other part of the program.

2.3. Validating higher-order contracts

In this section we give an intuitive explanation of how higher-order contracts are validated, a more operational description of this process is shown in Section 5. A higher-order contract is an agreement between two parties; the *server*

```
def notEmpty := contract: { |s| !s.isEmpty() };

def StackC := ObjectContract: {
 def push(o)  { int       -> void;};
 def pop()    { notEmpty  -> int; };
};

def contractedStack provide: stack withContract: StackC;
```

Fig. 5. Object contract definition and deployment.

and the *client*. The server is the provider of the initial contracted value (i.e. `map_pos`) and the client is its user context. Assigning blame in case of higher-order contracts is not as trivial as with first-order function contracts. Predicates defined over functions are in general undecidable and verifying them before starting the execution of the function under contract is in general impossible [24]. For this reason, checking the functional contracts of a higher-order contract must be delayed until these functions are used. It is easy to validate whether the array past as an argument to the `map_pos` function contains positive integers but validating the function past to the map function is in general undecidable and thus must be delayed until this function is used. Therefore, the execution point of a contract failure is dissociated from the execution point where the contracted function is initially applied.

This disassociation means that contract checking and determining blame assignment requires more than looking at the execution trace and blaming the last frame before the violation. When a contract is applied over a value it is decomposed into (smaller) contracts to protect the values that the server and the client exchange. In case of the `map_pos` function the functional contract defined over the function is taken out of the higher-order contract and applied over the function given as an argument. This function contract is a first-order function contract and can be validated as shown in Section 2.1. The contract system propagates a textual description of the provider and the server (blame labels) to the contract pieces it extracts form the contract in order to assign blame in case a violation is detected. However, it is important to realise that the provider of a higher-order argument is the client of the original contract and the user of the argument is the initially contracted function. Therefore, the contract system needs to swaps the roles of the parties and thus the blame labels. For arguments the client becomes the server and the server the client. For results the roles remain as before.

2.4. Object contract definition and deployment

Our approach for defining object contracts over prototypes is similar to how first-class class contracts are defined in Racket [27]. An object contract is defined by a protocol that is similar in syntax to an object definition. The defined methods in a protocol denote the minimal interface that an object has to implement. The body of these methods denotes the contract that is applied over the individual methods of the contracted object. It is often important to access the fields of the object in order to specify object contracts. Therefore, flat contracts over methods receive a reference to the contracted object as an optional second argument. As this argument is optional, flat contracts for functions can be reused in method contracts. Pre- and post-conditions that do not consume any arguments are created with the keyword function `contract:`.

An example of a contract for the elements, which can be pushed and popped off a stack is shown in Fig. 5. The object contract specifies that there should be at least two methods available namely `push` and `pop`. The contract defined over `push` states that the argument should be an integer. Similarly the values that are popped of the stack should be integers. The precondition of the pop operation states that the stack should not be empty. Object contracts are defined independently of the objects on which they are applied for two main reasons. First, the decoupled nature of defining object contracts has the advantage that the same contract can be reused for multiple objects. Second, defining the object contract independently of the contract on which it is deployed allows for better modularity.

With object contracts, we finish the overview of the basic contract system in AmbientTalk. In the next section, we present our extension to higher-order contracts, which allows the specification of contracts over the actual computation of a function while preserving support for higher-order programming.

3. Computational contracts

A computational contract is a higher-order contract over the execution of a contracted entity. Computational contracts are applicable over functions and objects and their higher-order arguments. In contrast to traditional higher-order contracts they are not restricted to only validate the interface of an applicable value but they can also express assertions over the computation that is associated with that value. Programmers can specify what has to happen or what should not happen during the execution of the computation.

As an example of a high-level computational contract 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, for example `positive` $\xrightarrow{!call(system.println)}$ `positive`. The contract defined over the `sqrt` function

```
def sqrt_c := provide: sqrt withContract:
positive -prohibit_c(system.println)-> positive;
```

Fig. 6. Using a prohibit contract to prevent the `sqrt` to apply system.println.

```
def map_pos := provide: map_pos withContract:
(positive -prohibit_c(system.println)-> positive) * arrayOf(positive)
-> arrayOf(positive);
```

Fig. 7. Defining a prohibit contract over the argument of the `map_pos` 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*(*system.println*), disallows all invocations of `system.println` during the execution of the `sqrt` function. The computational contract is active during the dynamic extent of the function application. Internally, the computational contract consist of two parts. First, the *interception component* is a description of *when* validation of the function under contract is needed, in our example when the method `system.println` is invoked. Second, the *blame assignment component* is applied when the interception component intercepts an interesting event. When the blame assignment component is applied it can decide to assign blame, update internal state and/or proceed with the computation, etc. The end programmer does not have to write these interception and blame assignment specification himself as the computational contract system provides high-level abstractions, which make it easy for the end programmer to specify temporal constraints.

Prohibition of certain function applications within the dynamic extent of a function application is only one type of computational contract. Specifying that a certain function application is mandatory or that a sequence of function calls should obey a certain protocol are two other examples, which are directly supported by the AmbientTalk computational contract system. To make it easy for the end programmer, all these variations can be expressed by two functions `ensure_c` and `prohibit_c`. These functions create low-level computational contracts. Depending on their arguments they either create a computational contract that prohibits or ensures that a certain function is applied or a protocol is followed during the execution of the contracted function. In the next sections we show how to define and use these type of high-level computational contracts in AmbientTalk.

3.1. 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(f)` prohibits the function `f` to be applied within the dynamic extent of the contracted function. Note that in this definition `f` denotes applications of the function that the variable `f` refers to when the contract is constructed.

In the rest of our examples all contracted functions are defined in a file called `"defs.at"` and exported with a contract. These functions are then imported and applied from a different module in a file called `"uses.at"`. In most examples there is no explicit module `"uses.at"` but when we show the AmbientTalk interaction prompt it is implicitly evaluated in this module.

Computational contracts are tightly integrated with the existing AmbientTalk contract system. For example the `prohibit_c` function can be used to contract exported functions in AmbientTalk as shown in Fig. 6. In this example[4] the function `sqrt` is exported with the prohibit contract that ensures that the function `system.println` is not applied. The prohibit contract assigns blame to the function `sqrt` whenever the function `system.println` 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 Fig. 7. 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 array `[1,2,3,4]` is shown below. The result of this function is as expected the array `[2,3,4,5]`.

```
Interactive AmbientTalk Shell, version 2.19 Contracts
>import ~/.defs;
>>nil
>map_pos({|x| x+1}, [1,2,3]);
>>[2, 3, 4]
```

[4] Note that there is a minus sign (`-`) in front of the `prohibit_c` contract. Besides representing that the computational contract is defined over the arrow, minus is actually a contract operator that binds the preconditions to the computational contract. Similarly, the -> operator binds the computational contract to the postconditions.

```
def display_average(a) {
 ((a.inject: 0 into: {|x,y| x+y})/a.length);
};

def moduleInterface := object: {
 def display_average := provide: display_average withContract:
                      arrayOf(pos) -ensure_c(system.println)-> any;
}
```

Fig. 8. Exporting the function `display-average` with a promise contract.

When the `map_pos` function is applied to a function that violates the computational contract, i.e. applies `system.println`, blame is assigned to the caller of the `map_pos` function. A transcript of this interaction is shown below. In the transcript there is no printout of the original array, instead an error message is presented. Contract checking of the computational contract stops the current evaluation and prevents the function `system.println` from being applied. The transcript shows that the violation was caused by the file `uses.at`. As the transcript was taken from the interaction window of the `uses.at` file, it can be easily deduced that the blame is assigned to the call made from the prompt. It is important to note that blame is assigned when the argument function is applied within the body of the `map_pos` function. When `map_pos` is applied it is 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. When a violation is detected during the execution of a function under contract, it is not always clear whether this is the fault of the caller or of the callee. Blame assignment solves this problem.

```
> map_pos( { |x| system.println(x); x+1; }, [1,2,3] )
```

```
1:8:uses.at violated the contract prohibit_c(system.println)
computational contract violation
origin:
at system.println(x)  (1:16:REPL)
at a.map:(f)  (44:17:defs.at)
at map_pos({ |x| system.println(x); x.+(1)}, [1, 2, 3])  (1:1:REPL)
```

To pinpoint the line that causes the violation of the computational contract the stack trace can be used (shown after the word `origin:`). It reveals that the origin of the violation is in the read–eval–print-loop on line 1, character 16. As shown in the stack trace it is that line that contains the call to the `system.println` function.

3.2. Ensure contracts

The dual of prohibiting an action is to *ensure* that an action is performed. An ensure contract 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`. Validating an ensure contract is more subtle than validating a prohibit contract as blame can only be assigned *after* the contracted function returns. Defining an ensure contract with computational contracts is as simple as defining a prohibit contract. Ensure contracts are created with the function `ensure_c`. When applied to the function `f` it returns an ensure contract that verifies that the function `f` is applied within the dynamic extent of the function under contract.

To exemplify the use of an ensure contract consider the function `display_average` shown in Fig. 8. This function takes an array of numbers and displays the average of the array. It is exported with an ensure contract that assigns blame when the promise of applying the function `system.println` is not held.

```
> display_average( [10,20] );
```

```
47:13:defs.at violated the contract ensure_c(system.println)
origin:
at display_average([10, 20])  (1:1:REPL)
```

This function `display_average` correctly computes the average of the list passed as an argument. Unfortunately, the function under contract does not apply the function `system.println`. Therefore applying the function `display_average` to the list `[10, 20]` leads to a violation of the ensure contract. A transcript of this example is shown above. As highlighted in the transcript the function `display_average` violates the `ensure_c(system.println)` contract.

3.3. Usage protocols

Computational contracts can also be used to validate that a usage protocol is respected. Until now we have only considered computational contracts where the application of a single function is validated. In this section, we show computational contracts where the programmer describes a partial specification of the path of applications a certain function

```
def OpenCloseProtocol() {
 UsageProtocol:  {
  def start()    { closed(); };
  def closed()   { (on:  openFile)  => { opened(); }};
  def opened()   { (on: closeFile)  => { end();    }};
  def end()      { (on: anyMethod)        => { false;    }};
 };
};
```

Fig. 9. Defining a protocol for opening and closing files.

```
def readCharFromFile(s) {
 openFile(s).readChar();
};

def moduleInterface := object: {
 def readCharFromFile  := provide: readCharFromFile withContract:
 string -ensure_c(OpenCloseProtocol)-> char;
}
```

Fig. 10. Defining a protocol contract over the `readCharFromFile` function.

should or should not follow. In our implementation this path is expressed by a finite state machine.[5] As shown in Fig. 9, this finite state machine describes those functions that can be applied successively. The start state of the finite state machine indicates that the finite state machine has to be initialised to the state closed. Besides the start state the finite state machine has three states: `closed`, `opened`, `end`. In each of these states there is one possible transition, for example when the finite state is in the closed state, function invocations to the function `openFile` will transition the finite state machine to the opened state.

Each state can have multiple transitions and are denoted with the following syntax `(on: f)=> B;` where `f` refers to a function and `B` is a transition function expressed in AmbientTalk code. This transition function is applied to the arguments of the intercepted function. If multiple transitions are defined the first applicable transition function is chosen. The result of evaluating the transition function `B` is used as the next state of the finite state machine. In case the returned value is `false`, a contract violation is detected. States can also receive arguments which can be used in the transition code in order to decide to which state to transition next.

The protocol specifies that the `end` state can be reached after exactly one application of the function `openFile` followed by one application of the function `closeFile`. Functions that are not specified in the protocol can always be applied. For example the application sequence `openFile`, `system.println`, `closeFile` leads to the end state. In the `end` state the special wildcard qualifier `anyMethod` is used. This means that, calling any function mentioned in the usage protocol while in the `end` state results in a violation of the protocol.

When an application sequence does not follow the protocol blame is assigned. For example, the application sequence `openFile`, `closeFile`, `openFile` is not allowed as the `end` state does not allow any applications to `openFile`.

Ensure protocols. Once a protocol is defined it can be used to create a computational contract with the `ensure_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 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 `OpenCloseProtocol` consider the `readCharFromFile` function shown in Fig. 10. This function opens a file and reads one character from this file. The `readCharFromFile` function is exported with a contract that ensures that the `OpenCloseProtocol` 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 `readCharFromFile` function results in an error message as shown below. In the error message we see that blame is assigned to the file `defs.at`. It is also specified that the violation was a `ensure_protocol_c` contract violation.

```
> readCharFromFile("testFile.at");
```

```
52:13:defs.at violated the contract ensure_protocol_c
origin:
at readCharFromFile("testFile.at")  (1:1:REPL)
```

[5] Our implementation of the finite state machine is in the line of [20].

```
def createWindowProtocol :=
  UsageProtocol: {
    def start() {  checkWindow(2) };
    def checkWindow(x) {
      (on: createWindow) =>  {
        if: ( x == 1) then: { checkWindow(0); } else: { end(); }
      };
   };
  };
```

Fig. 11. Open windows protocol definition.

```
def readAndShow(filename) {
   ...
   createWindow( ... );
   createWindow( ... );
   ...
}

def readAndShow := provide: readAndShow withContract:
                   string -prohibit_c(create-window-protocol)-> any);
```

Fig. 12. Defining a prohibit protocol contract over the `readAndShow` function.

Prohibit protocols. A computational contract that prohibits following a protocol can be created with the function `prohibit_c`. Such a computational contract assigns blame 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 Fig. 11. This protocol reaches the end-state after exactly two applications of the `createWindow` 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.

The function `readAndShow` shown in Fig. 12, 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 (createWindowProtocol)`. Using the function `make-and-show-window` leads to a violation of the prohibit contract.

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. Computational contracts for object protocols

In this section we show the use of computational contracts in order to restrict temporal orderings of method calls on an object. Such temporal orderings over method calls are known as *object protocols* and are an active point of research. Recently, the use of object protocols has been analysed in a number of open-source projects, comprising almost two million lines of code. A remarkable result from this study is that about 90% of the protocols found, fit into one of five categories [2]. In this section we show the applicability of computational contracts for object protocols by defining abstractions that implement the categories as presented in [2].

4.1. Initialisation

The first object protocol category concerns the initialisation of objects. In certain situations an objects must be initialised *after* construction time but before the object is meant to be used. An example of this category can be found in the `AlgorithmParameters` class of Java, only after one of its three init methods has been invoked calls to `getEncoded` are allowed. In the initialisation category, calls to an instance method *m* after construction-time will result in an error unless an initialising method *i* has been called at least once before [2]. The implementation of this object-protocol in AmbientTalk is shown in Fig. 13. Method names in the object protocol can be matched based on regular expressions, i.e. to select all methods that start with `set` the following regular expression can be used `"set.*"`. In order to use the `InitialisationProtocol`, the programmer thus only needs to provide the regular expression that selects the initialisation methods. An example where the `InitialisationProtocol` will only allow method invocations to the contracted object after a method invocation to the method `initWithPredicate` is shown at the bottom of Fig. 13. A simple variation on the `InitialisationProtocol` where calls to the `init` method are only allowed once, only requires adding another case in the `endState`, i.e. `(on: initMethod) => {false};`

```
def InitialisationProtocol(initMethod) {
  ObjectProtocol: {
    def startState()    { (on: initMethod) =>  {  endState() };
                          (on: anyMethod ) =>  {  false      }; };
    def endState()      { (on: anyMethod ) =>  {  endState() }; };
    def start()         {  startState(); };
  };
};
...
provide: filter withContract:
ensure_c( InitialisationProtocol("initWithPredicate") );
```

Fig. 13. Initialisation Protocol.

```
def TypeQualifierProtocol(initMethod,predicate, disallowed) {
 ObjectProtocol: {
  def startState() {
   (on: initMethod) =>  { |arg|
    if: !predicate(arg) then: {
      allAllowed();
    } else: {
     disallowedState();
    };
   };
   (on: anyMethod)  => {  startState(); };
  };

  def disallowedState() {
   (on: disallowed) => {  false; };
   (on: anyMethod)  => {  disallowedState(); };
  };

  def allAllowed() {
   (on: anyMethod)  => { allAllowed(); };
  };

  def start() {
   startState();
  };
 };
};
```

Fig. 14. TypeQualifierProtocol.

4.2. Deactivation

The deactivation object protocol category verifies the deactivation of an object. After deactivation, any method invocation on a deactivated instance results into an exception. In the deactivation category, calls to an instance method *m* will fail after some method *d* is called on the same instance, and it will always fail for the rest of the object's lifetime [2]. Like initialisation, object protocols may or may not permit *d* to be called more than once. The implementation of the deactivation protocol is very similar to the initialisation protocol and thus omitted from this paper.

4.3. Type qualifier

Some types disable certain methods for the lifetime of the object. In the type qualifier category, an object instance will enter an abstract state *S* at construction-time, which it will never leave [2]. Calls to an instance method *m*, if it is disabled in state *S* will always fail. In many cases the abstract state that newly constructed instances inhabit can be set by parameters to the constructor. This case is shown in Fig. 14. This object-protocol is instantiated with an initialiser method, a predicate and a regular expression describing the disallowed methods. When the `initMethod` is invoked in the `startState` the predicate will be applied to the argument passed to the initMethod. This argument is accessible from within the protocol and bound to the `arg` variable. Depending on the predicate, the state machine will transition to the `allAllowed` or `disallowedState`.

4.4. Dynamic preparation

In the dynamic preparation category, an instance method *m* will fail unless another instance method *p* is called before it [2]. If we think of types in this category as having two states, ready and not-ready, this category is distinguished from the initialisation category in that an object may dynamically change from ready to not-ready at numerous points in its lifetime (i.e. it is not monotonic). Implementing this category boils down to adding transition relations that will move the object-protocol from the not-ready state to the ready state when certain methods are applied. In the not-ready state invocation to the disallowed method violate the object-protocol.

4.5. Redundant operation

In the redundant operation category, a method *m* will fail if it is called more than once on a given instance [2]. The implementation of this abstraction is omitted from the paper, as it is very similar to the `createWindowProtocol` shown in Fig. 11.

In this section we showed the applicability of computational contracts for object-protocols by implementation object-protocol categories. These categories account for 90% of the object protocols currently found in the wild. While the implementation of these object protocols with computational contracts seems trivial, current implementation of object-protocols are low level and scattered throughout the code. With computational contracts we offer direct support for object-protocols. Further the use of computational contracts for the implementation of object-protocols has the advantage that the validation code can be separated from implementation. Direct support for object-protocols also has the advantage that the programmer will be able to define more complex object-protocols. Finally, by making use of our contract system the programmer receives blame assignment information about where the violation against the object-protocol takes place and the module that is responsible for the violation.

5. Operational semantics of computational contracts

In this section we precisely describe the inner working of the computational contract system by presenting an operational semantics. In order to concentrate on the specific mechanism of checking computational contracts, we describe their operational semantics in a simple higher-order functional language à la Scheme. The operational semantics of the full AmbientTalk language are described elsewhere [30]; the integration of computational contracts in AT does not raise any specific issue beyond the mechanism described here. An executable specification of the semantics is also available for PLT Redex [13].[6] The figures of the specification as explained here are generated directly from the PLT Redex semantics. We start our explanation by describing the CEK model of a Scheme like language, and then explain how to extend it to support computational contracts.

5.1. CEK model and syntax definition

We first show a core Scheme like language with syntax for computational contracts and incrementally show the extensions needed to define computational contracts. The semantics is defined as a variation on a CEK machine. The CEK machine defines program behaviour by defining transition relations from one program state to the next. The representation of a program state consists of the control string together with its environment and the continuation of the computation. In order to express computational contracts next to a variable environment also a contract environment is needed. Formally the state of a computation consists of a tuple:

1. The control string (C), the environment (E) and the contract environment (CE).
2. The continuation code (K).

Environments are finite maps for the set of variables x to the set of values v. If E is an environment then $E[x := v]$ is like E except on the point x where it is v. Reduction rules of the machine are written in the form $\langle\langle \mathrm{C\ E\ CE}\rangle\ \mathrm{K}\rangle \Rightarrow \langle\langle \mathrm{C'\ E'\ CE'}\rangle\ \mathrm{K'}\rangle$. The contract environment *CE* is an *ordered* list of prohibit and ensure contracts.

Fig. 15 shows the core syntax of the λ_c (c for contract) language. λ_c is a simple Scheme like language with booleans, numbers and lists together with the primitive operations applicable to them. The basic expressions consists of values, variables, function applications, if statements, and assignment. There are three syntactical expression for the definition of contracts. Flat contracts are represented by (`flat` e) where e is expected to evaluate to a predicate function. For example, the definition of a flat contract that verifies that a value has to be bigger than `10` can be represented as follows: (`flat` (λ (x) ($>$ x 10))). Composing flat contracts in order to define a functional contract is done by using the arrow operator, ($\rightarrow$ c_a c_r). In this representation c_a is the contract defined over the arguments of the function under contract and c_r the

[6] http://soft.vub.ac.be/~cfscholl/index.php?page=at_contracts.

$$
\begin{array}{rl}
v & ::= (\lambda\ (x)\ e) \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{empty} \mid \mathit{number} \mid (\mathsf{cons}\ \mathit{VC}\ \mathit{VC}) \\
e & ::= v \mid x \mid (e\ e) \mid (o\ e\ \ldots) \mid (\mathsf{if}\ e\ e\ e) \mid (\mathsf{mon}\ x\ x\ c\ e) \mid (\mathsf{blame}\ x) \mid (\mathsf{app/prim}\ e\ e) \\
o & ::= \mathsf{eq?} \mid \mathsf{cons} \mid \mathsf{first} \mid \mathsf{rest} \mid \mathsf{empty?} \mid > \mid < \\
c & ::= (\mathsf{flat}\ e) \mid (\rightarrow\ c\ c) \mid (cc\ e) \\
cc & ::= \mathsf{prohibit} \mid \mathsf{ensure} \\
\mathit{end} & ::= (\mathsf{term}\ v) \\
\mathit{MC} & ::= [e\ E\ CE] \\
\mathit{VC} & ::= [v\ E\ CE] \\
K & ::= \mathsf{stop} \mid [\mathsf{ifk}\ \mathit{MC}\ \mathit{MC}\ K] \mid [\mathsf{app1\text{-}k}\ \mathit{MC}\ E\ CE\ K] \mid [\mathsf{app2\text{-}k}\ \mathit{VC}\ E\ CE\ K] \\
& \quad \mid [\mathsf{appprim1\text{-}k}\ \mathit{MC}\ CE\ K] \mid [\mathsf{appprim2\text{-}k}\ \mathit{VC}\ CE\ K] \mid [\mathsf{op\text{-}k}\ o\ E\ CE\ e\ K] \\
& \quad \mid [\mathsf{op\text{-}k2}\ o\ \mathit{VC}\ K] \mid [\mathsf{chk}\ E\ CE\ x\ x\ c\ K] \mid [\mathsf{chk\text{-}comp}\ cc\ x\ \mathit{VC}\ K] \\
& \quad \mid [\mathsf{chk\text{-}ensure}\ \mathit{CE}_{\mathit{fun}}\ K] \\
E & ::= ((x\ \mathit{MC})\ \ldots) \\
\mathit{CE} & ::= ((cc\ \mathit{VC}\ x)\ \ldots) \\
x & ::= \mathit{variable\text{-}not\text{-}otherwise\text{-}mentioned}
\end{array}
$$

Fig. 15. Core syntax of the λ_c language.

$$
\begin{array}{rcll}
e & \longrightarrow & [[e\ E_0\ EC_0]\ \mathsf{stop}] & \text{[INIT]} \\
[[v\ E\ CE]\ \mathsf{stop}] & \longrightarrow & v & \text{[TERM]} \\
[[x_c\ E\ CE]\ K] & \longrightarrow & [E\ [x_c]\ K] & \text{[VAR]} \\
[[(\mathsf{if}\ e\ e_1\ e_2)\ E\ CE]\ K] & \longrightarrow & [[e\ E\ CE]\ [\mathsf{ifk}\ [e_1\ E\ CE]\ [e_2\ E\ CE]\ K]] & \text{[IF]} \\
[[\mathsf{true}\ E\ CE]\ [\mathsf{ifk}\ \mathit{MC}_1\ \mathit{MC}_2\ K]] & \longrightarrow & [\mathit{MC}_1\ K] & \text{[IF/TRUE]} \\
[[\mathsf{false}\ E\ CE]\ [\mathsf{ifk}\ \mathit{MC}_1\ \mathit{MC}_2\ K]] & \longrightarrow & [\mathit{MC}_2\ K] & \text{[IF/FALSE]} \\
[[(\mathsf{mon}\ x_h\ x_f\ c\ e)\ E\ CE]\ \ K] & \longrightarrow & [[e\ E\ CE]\ [\mathsf{chk}\ E\ CE\ x_h\ x_f\ c\ K]] & \text{[MON]} \\
[VC\ (\mathsf{chk}\ E\ CE\ x_h\ x_f\ \ (\mathsf{flat}\ e)\ K)] & \longrightarrow & [[(\mathsf{if}\ (e\ \mathsf{cv})\ \mathsf{cv}\ (\mathsf{blame}\ x_h))\ E\ [\mathsf{cv}{:=}\mathit{VC}]\ CE]\ K] & \text{[FLAT]} \\
[VC\ (\mathsf{chk}\ E\ CE\ x_h\ x_f\ \ (\rightarrow\ c_1\ c_2)\ K)] & \longrightarrow & [[(\lambda\ (\mathsf{p})\ (\mathsf{mon}\ x_h\ x_f\ c_2\ (\mathsf{org}\ (\mathsf{mon}\ x_f\ x_h\ c_1\ \mathsf{p}))))\ E\ [\mathsf{org}{:=}\mathit{VC}]\ CE]\ K] & \text{[HO]} \\
[VC\ [\mathsf{chk}\ E\ CE\ x_h\ x_f\ [cc\ e]\ K]] & \longrightarrow & [[e\ E\ CE]\ [\mathsf{chk\text{-}comp}\ cc\ x_h\ \mathit{VC}\ K]] & \text{[CC-ARG]} \\
[VC\ [\mathsf{chk\text{-}comp}\ cc\ x_h\ [v\ E\ CE]\ K]] & \longrightarrow & [[v\ E\ (CE \cup (cc\ \mathit{VC}\ x_h))]\ K] & \text{[CC]} \\
[[(e_{\mathit{fun}}\ e_{\mathit{arg}})\ E\ CE]\ K] & \longrightarrow & [[e_{\mathit{fun}}\ E\ CE]\ [\mathsf{appprim1\text{-}k}\ [e_{\mathit{arg}}\ E\ CE]\ CE\ K]] & \text{[APP/PRIM-FUN]} \\
[VC_{\mathit{fun}}\ [\mathsf{appprim1\text{-}k}\ \mathit{MC}_{\mathit{arg}}\ CE_{\mathit{app}}\ K]] & \longrightarrow & [\mathit{MC}_{\mathit{arg}}\ [\mathsf{appprim2\text{-}k}\ VC_{\mathit{fun}}\ (CE_{\mathit{app}} \setminus VC_{\mathit{fun}})\ \ K]] \quad \text{where } VC_{\mathit{fun}} \notin CE_{\mathit{app}} & \text{[APP/PRIM-ARG]} \\
[VC_{\mathit{fun}}\ [\mathsf{appprim1\text{-}k}\ \mathit{MC}_{\mathit{arg}}\ \ CE_{\mathit{app}}\ \ K]] & \longrightarrow & (\mathsf{blame}\ x_h) \quad \text{where } CE_{\mathit{app}}\ [VC_{\mathit{fun}}]{=}x_h & \text{[APP/PRIM-ARG-BLAME]} \\
[VC_{\mathit{arg}}\ [\mathsf{appprim2\text{-}k}\ [(\lambda\ (x)\ e_{\mathit{body}})\ E_{\mathit{fun}}\ CE_{\mathit{fun}}]\ CE_{\mathit{app}}\ K]] & \longrightarrow & [[e_{\mathit{body}}\ E_{\mathit{fun}}\ [x{:=}VC_{\mathit{arg}}]\ (CE_{\mathit{fun}} \cup CE_{\mathit{app}})]\ [\mathsf{chk\text{-}ensure}\ CE_{\mathit{fun}}\ K]] & \text{[APP/PRIM-BODY]} \\
[[v\ E\ CE_{\mathit{app}}]\ [\mathsf{chk\text{-}ensure}\ CE_{\mathit{fun}}\ K]] & \longrightarrow & [[v\ E\ CE_{\mathit{app}}]\ K] \quad \text{where } CE_{\mathit{app}} \Cap CE_{\mathit{fun}}{=}\emptyset & \text{[CHK-ENSURE]} \\
[[v\ E\ CE_{\mathit{app}}]\ [\mathsf{chk\text{-}ensure}\ CE_{\mathit{fun}}\ K]] & \longrightarrow & (\mathsf{blame}\ x_h) \quad \text{where } CE_{\mathit{app}} \Cap CE_{\mathit{fun}}{=}x_h & \text{[CHK-ENSURE-BLAME]}
\end{array}
$$

Fig. 16. Basic reduction rules of the CEK machine for computational contacts.

contract over the return value of the function under contract. Note that the definition of contracts is recursive, therefore both c_a and c_r can be functional contracts.

Computational contracts in the semantics are limited to `prohibit` and `ensure` contracts, both take an expression that is expected to evaluate to a function. Similar to the approach by Dimoulas et al. [7] guarding an expression with a contract is expressed by a monitor construct ($\mathtt{mon}_i^j\ c\ e$). The two labels i, j respectively indicate the provider of the expression and the user of the expression. Finally, blame is expressed by ($\mathtt{blame}\ i\ j$).

CEK-machine reductions rules. The rules that govern the CEK machine are shown in Fig. 16. There are four groups of rules, a first group governs initialisation and termination, a second governs `if` statements, the third group deals with function applications and the last group deals with higher-order contracts.

Initialisation and termination.

- INIT: In order to evaluate a program e, the machine starts with the initial environment E_0 and the stop continuation `stop`. In the initial environment E_0 there are no bindings and the set of variables are all mapped to error i.e. $E_0 \equiv x \mapsto \mathtt{error}$ for all variables x. The initial contract environment CE_0 is empty. After initialisation the machine steps through a number of reductions until the machine reaches a terminal state.
- TERM: When a terminal state is reached the machine stops and the final value v is returned as the answer of the evaluation.

If statements. The evaluation of `if` expressions is governed by IF, IF/TRUE and IF/FALSE and is completely standard.

Higher-order contracts. The evaluation of higher-order contracts consists of three rules.

- MON: The monitor reduction rule moves evaluation to the body of the expression under contract and remembers the contract that is defined over the value.
- FLAT: When the expression is evaluated to a value and there was a flat contract defined over this value the `flat` contract is transformed into a conditional that tests whether the predicate of the flat contract is obeyed.
- HO: Higher-order contracts are recursively translated into a wrapper function, which monitors the argument and result of the original function. Note that the blame labels are switched for the argument monitor as explained in Section 2.3.

Computational contracts:. The *definition* of computational contracts consists of three rules. Validating computational contracts is interwoven with the reduction rules for function applications.

- CC-ARG: A `ensure` or `prohibit` computational contract takes an expression *e* that is expected to evaluate to a function. The CC-ARG rule evaluates the expression *e* of the computational contract and creates a new continuation to be applied over the monitored value.
- CC: Applying the contract over a value creates a new value on the top of the stack that is identical to the original value but the contract environment *EC* of this new value is extended with the evaluated computational contract.

Function applications. The evaluation of function applications consists of the following rules and are interwoven with the contract checking semantics.

- APP/PRIM-FUN: This rule moves evaluation to the applied function.
- APP/PRIM-ARG: This rules evaluates the argument expression of the function application. Note that the function that is being applied cannot be contained in the prohibit list of the contract environment. Finally when a function is applied the matching ensure contracts are removed from the contract environment (i.e. $CE_{app} \setminus VC_{fun}$).
- APP/PRIM-BLAME: When a function is being applied that is prohibited blame needs to be assigned. This is done by traversing the list of prohibited functions from the oldest prohibited function till the newest prohibited function. In the reduction rules this is expressed as $CE_{app}[VC_{fun}] = x_h$ where x_h is the blame label corresponding to the prohibit contract.
- APP/PRIM-BODY: After the argument is evaluated the body of the function is evaluated in the application environment extended with the binding of the formal parameter to the actual parameter.
- CHK-ENSURE and CHK-ENSURE-BLAME: When the function application is evaluated to a value. The check ensure continuation validates whether all the ensure contracts of the applied function are removed from the contract environment (with the $\Cap$ operator). In case a remaining ensure contract is found in the environment blame is assigned.

6. Implementation of computational contracts in AmbientTalk/C

The implementation of computational contracts in AmbientTalk/C is based on aspect-oriented programming. Therefore, we first give a brief overview of the aspect-oriented language constructs and then go into detail of how computational contracts can be implemented with those constructs.

6.1. Aspect-oriented language extensions

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 programmer specifies when the advice has to be applied by giving a *point cut* . Whenever the join point descriptor matches a join point the advice is executed.

An aspect consists of a pointcut descriptor and a corresponding advice. The advice is executed whenever the pointcut descriptor matches the current join point stack. A point cut is represented by a predicate function which is applied to a stack of join points. When the aspect descriptor returns `true` the corresponding advice is executed. The programmer can alter the execution of the program by executing other functionality before, after or instead of the intercepted function application. Advice is implemented as a closure that is applied by the underlying system with a function *p*. This function *p* allows the programmer to continue with the originally intercepted function application. The advice is expected to return a function that is applied by the underlying system to the arguments of the intercepted function application.

Now that it is clear how an aspect can be defined, there is still the question of how to deploy an aspect and how to limit its scope. Moreover, there also is the question whether the aspect should be active in the static or dynamic scope of evaluation. Full fledged aspect languages have constructs for both dynamic and static aspect deployment [11]. As the semantics of computational contracts depends only on dynamically deployed aspects we limit the rest of the explanation to the specification of dynamic aspects. In AmbientTalk/C, `fluid: A deploy: B`, deploys an aspect `A` in the dynamic extent of executing the body `B`.

```
def cc(pc, adv, dom, rng) {
 {|pos, neg|
  { |val|
   if: (is: val taggedAs: Closure) then:  {
    { |x|
     def verified := dom(neg, pos)(x);
     def adv := adv(pos,neg);
     def asp := aspect: pc advice: { |proceed, args|
      adv(proceed,args);
     };
     def result := nil;
     fluid: asp deploy: {
      result := val(verified);
     };
     rng(pos,neg)( result);
    }
   } else: {
    blame(pos);
   };
  };
 };
};

def prohibit_c(pc) {
 cc(pc,
  {|pos,neg| {|p,a| blame(pos)}},
  flat( {|x| true;}),
  flat( {|x| true;}))
};

def ensure_c(pc) {
 def called := false;
 cc(pc,
  {|pos,neg| {|p,a| called := true; p(a); }},
  flat( {|x| called := false; true;}),
  flat( {|x| called;} ) );
}
```

Fig. 17. High-level abstraction for computational contracts.

6.2. Verification and blame assignment of computational contracts

We now show a didactical implementation[7] of computational contracts based on aspects and explain how blame assignment works.

At the top of Fig. 17, the computational contract constructor function `cc` is shown. This function is used to create a computational contract (i.e. it implements the (-) and -> operators). The first argument of this function is the interception component of the computational contract. The interception component specifies the exact points where the contract needs to be validated. The second argument is the blame assignment component and is almost the same as an advice. The difference is that a blame assignment component also receives blame assignment labels. The two last arguments are the domain contract and the range contract. A computational contract verifies its contracted function very similarly to the way higher-order contracts are verified in Section 5. First, the domain contract is verified (line 6). Second, the blame assignment component is initialised and an aspect is created (line 7–10). This aspect is used to intercept interesting events in the dynamic extent of the applied contracted function Subsequently, this aspect is deployed and the function over which the contract is defined is applied (12–14). The blame assignment component is applied when a matching join point is encountered in the dynamic extent of applying the contracted function, as specified in the advice (line 9). *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 contract.*

The definition of a prohibit and ensure contracts can now be defined in terms of our general computational contract definition, as is shown in Fig. 17. A prohibit contract simply assigns blame in case the blame assignment component is applied. An ensure contract initialises a variable *called* to be *false* in the precondition and sets this variable to *true* in the blame assignment component. The postcondition returns this variable and blame is assigned when the variable was not set to *true*.

7. Discussion

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

[7] Available online: http://soft.vub.ac.be/~cfscholl/AmbientTalk/ambienttalk.html.

```
def  process( f) {  ... (f 4) ... }
def  remove(path) { ... }

def moduleInterface := object: {
 def process := provide: process withContract: any -prohibit_c(remove)-> any;
 def remove := provide: remove withContract: string -> bool;
};
```

Fig. 18. Example where a computational contract and a pre/post contract can be active over the *same* function.

7.1. Blame precedence

Note that a function can be subject to validation by multiple computational contracts at the same time. In our implementation we always give precedence to the oldest deployed computational contract. The reasoning behind this decision is that in case of multiple *identical* contracts the oldest deployed contract indicates the first provider of the value who promised the contract. Successive identical contract applications mean that a module has passed a contracted value to another module that expects the same contract. If in this new module a violation is detected blame is assigned to the module who first promised the contract.

It is also possible that a function is subject to validation by both a computational contract and a pre/post contract at the same time. In that case, the contract system should give precedence to either the computational contract or the pre/post contract.

To clarify this, consider the code example show in Fig. 18. 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 hard-disk. 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 the difference that the contract is validated. Applying the exported function `process` to the exported function `remove` results in the computational contract to be validated. 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. Note that the function `remove` is still contracted by the pre/post contract `(string? → boolean?)`. At the same time a computational contract that prohibits applications of the function `remove` is also active. 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 hard-disk 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 a semantic `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 pointer equality function `eq`. In that case, the computational contract is not applied before the pre/post contract; hence the programmer gets a precondition violation.

7.2. 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 [14] 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 [4]. 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. [8] 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 validation of a computational contract the pre/post contract might violate the computational contract. Imagine a contracted function with

a flat contract `printArgument`, which allows any argument to pass but also `prints` the argument. When this function is applied a computational contract that disallows any application of the `system.println` function can be active. 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.

8. Related work

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. [17] and Holland [18] 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) [26]. They define a higher-order method as any method whose behaviour critically depends on one or more mandatory calls. This approach does not support higher-order contracts, i.e contracts over the argument values cannot be specified. The verification is based on model contracts and violations are only found when the model program can be matched with the body of the contracted method. Violations against the contract in the dynamic extend of the contracted method (i.e. calls to other methods within the body of the contracted method) are not validated.

A very related approach by Fischer [15] introduces trace-based assertions. These are similar in nature to the protocol contracts as shown in Section 3.3. However, trace-based assertions do not support functional contracts to be defined over the argument values of a function. A related approach by Soundarajan and Tyler [29] allows trace-based specifications by means of hook methods that extend the trace. This system suffers from similar limitations as Fischer's system with respect to contracts over argument values.

In computational contracts an object contract is applied at the moment an object crosses the module boundaries. In JML like languages contract definition and class definition are intertwined. Therefore viewing objects of a class under different contracts requires the programmer to add new subclasses to the class hierarchy which leads to a lot of code repetition. In AmbientTalk and other prototype based languages such as JavaScript, the object system is classless and objects are created ex-nihilo. This difference in the place where the contract is defined was already observed in the context of higher-order object-oriented contracts by Strickland et al. [27]. Our system brings the power of temporal contracts similar to JML into higher-order dynamically-typed languages without leading to code repetition.

In static type systems, several proposals [5,12] have focused on higher-order function verification, to be sound they introduce heavy restrictions on the scoping mechanism.

In Typestate oriented programming [28] the programmer defines an object which has multiple states, transition from one state to another is linked to certain method invocations. This is very similar to the object protocols shown in Section 4. However, the `ensure` and `prohibit` protocols expressible with computational contracts can transition from one state to another by arbitrary method invocations (not limited to a single object).

MaC [9] is a runtime validation system where program executions points, such as the application of a function, are reified as events. Over these events the programmer can write rules for validating 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 remarkable contract system that goes beyond pre- and post-conditions was recently proposed by Heidegger et al. [16]. 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 focus on checking the *access* to certain variables and do not check function invocations. They can be viewed as a particular instantiation of computational contracts and we plan to implement them in our framework as future work.

As pointed out throughout the paper computational contracts are an extension to the work on higher-order contracts [14,4,27,8,7]. Most higher-order contract frameworks have focused on blame assignment and do not provide abstractions in order to define temporal properties. Higher Order Temporal (HOT) Contracts [10] extend prior higher-order contract systems to also express and validate 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 nor external module calls. This makes that HOT contract differ from computational contracts in two ways. First, the `prohibit_c` contract shown in Fig. 6 and similar contracts that refer to system calls are not easily expressible with a HOT contract because external calls to the system module are not in the trace. It is possible to define a separate module that wraps the system library into a different module with an appropriate temporal higher-order contract but it would not be trivial. Moreover, users of the original module cannot see this contract with the wrapped system module at the interface level. Second, 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 validate 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 and external applications can be monitored.

9. 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 validate 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 validate 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 *ensure* and *prohibit* computational contracts respectively.

References

[1] M. Barnett, K.R.M. Leino, W. Schulte, The Spec# Programming System: An Overview, Springer, 2004, pp. 49–69.
[2] N.E. Beckman, D. Kim, J. Aldrich, An empirical study of object protocols in the wild, in: Proceedings of the 25th European Conference on Object-oriented Programming, ECOOP, July 2011, pp. 2–26.
[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, D. Watkins, Making components contract aware, Computer 32 (7) (1999) 38–45. ISSN 0018-9162.
[4] M. Blume, D. McAllester, Sound and complete models of contracts, Journal of Functional Programming 16 (2006) 375–414.
[5] W. Damm, B. Josko, A sound and relatively complete hoare-logic for a language with higher type procedures, Acta Informatica 20 (1983) 59–101.
[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter, Ambient-oriented programming in ambienttalk, in: D. Thomas (Ed.), Proceedings of the 20th European Conference on Object-oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 4067, Springer, 2006, pp. 230–254. http://dx.doi.org/10.1007/11785477_16. ISBN 978-3-540-35726-1.
[7] C. Dimoulas, M. Felleisen, On contract satisfaction in a higher-order world, ACM Transactions on Programming Languages and Systems 33 (5) (2011) 16:1–16:29. ISSN 0164-0925.
[8] C. Dimoulas, R.B. Findler, C. Flanagan, M. Felleisen, Correct blame for contracts: no more scapegoating, in: Proceedings of the 38th Annual Symposium on Principles of Programming Languages, POPL, 2011, pp. 215–226.
[9] N. Dinesh, A. Joshi, I. Lee, 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, 2008, pp. 110–124.
[10] T. Disney, C. Flanagan, J. McCarthy, Temporal higher-order contracts, in: International Conference on Functional Programming, 2011, pp. 176–188.
[11] C. Dutchyn, D.B. Tucker, S. Krishnamurthi, Semantics and scoping of aspects in higher-order languages, Science of Computer Programming 63 (3) (2006) 207–239. ISSN 0167-6423.
[12] G.W. Ernst, J.K. Navlakha, W.F. Ogden, Verification of programs with procedure-type parameters, Acta Informatica 18 (1982) 149–169.
[13] M. Felleisen, R.B. Findler, M. Flatt, Semantics Engineering with PLT Redex, first ed., The MIT Press, 2009, ISBN 0262062755, 9780262062756.
[14] R.B. Findler, M. Felleisen, Contracts for higher-order functions, in: Proceedings of the Seventh International Conference on Functional Programming, ICFP, 2002, pp. 48–59.
[15] C. Fischer, Combination and implementation of processes and data: from CSP-OZ to Java, Ph.D. Thesis, University of Oldenburg, Germany, Jan. 2000.
[16] P. Heidegger, A. Bieniusa, P. Thiemann, Access Permission Contracts for Scripting Languages, POPL'12, Jan. 2012, pp. 111–122.
[17] R. Helm, I.M. Holland, 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, 1990, pp. 169–180.
[18] I.M. Holland, Specifying reusable components using contracts, in: European Conference on Object-Oriented Programming, ECOOP, 1992, pp. 287–308.
[19] M. Karaorman, U. Holzle, J. Bruno, jcontractor: A reflective Java Library to Support Design by Contract, Technical Report, Santa Barbara, CA, USA, 1999.
[20] S. Krishnamurthi, Educational pearl: automata via macros, Journal of Functional Programming 16 (3) (2006) 253–267.
[21] B. Meyer, Eiffel: The Language, in: Prentice Hall Object-Oriented Series, 1991, ISBN 0132479257.
[22] B. Meyer, Applying design by contract, Computer 25 (1992) 40–51.
[23] R. PlŽsch, J. Pichler, Contracts: from analysis to C++ implementation, in: Technology of Object-oriented Languages and Systems, IEEE Computer Society, 1999, pp. 248–257.
[24] H.G. Rice, Classes of recursively enumerable sets and their decision problems, Transactions of the American Mathematical Society 74 (2) (1953) 358–366.
[25] C. Scholliers, E. Tanter, W. De Meuter, Computational contracts, in: Workshop on Scheme and Functional Programming, 2011.
[26] S.M. Shaner, G.T. Leavens, D.A. Naumann, Modular verification of higher-order methods with mandatory calls specified by model programs, in: Proceedings of the 22nd Annual Conference on Object-oriented Programming Systems and Applications, OOPSLA'07, 2007, pp. 351–368.
[27] T.S. Strickland, M. Felleisen, Contracts for first-class classes, in: Proceedings of the 6th Symposium on Dynamic languages, DLS, 2010, pp. 97–112.
[28] J. Sunshine, K. Naden, S. Stork, J. Aldrich, É. Tanter, First-class state change in plaid, in: Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2011, pp. 713–732.
[29] B. Tyler, N. Soundarajan, Black-box testing of grey-box behavior, in: FATES'03, 2003, pp. 1–14.
[30] T. Van Cutsem, C. Scholliers, D. Harnie, W. De Meuter, An Operational Semantics of Event Loop Concurrency in AmbientTalk, Technical Report VUB-SOFT-TR-12-04, Vrije Universiteit Brussel, 2012.