Functional Testing in the Focal environment

Matthieu Carlier, Catherine Dubois

CÉDRIC-ENSIIE,
1 square de la résistance, 91025 Évry Cedex, France,
{carlier, dubois}@ensiie.fr

Abstract. This article presents the generation and test case execution under the framework Focal. In the programming language Focal, all properties of the program are written within the source code. These properties are considered, here, as the program specification. We are interested in testing the code against these properties. Testing a property is split in two stages. First, the property is cut out in several elementary properties. An elementary property is a tuple composed of some pre-conditions and a conclusion. Lastly, each elementary property is tested separately. The pre-conditions are used to generate and select the test cases randomly. The conclusion allows us to compute the verdict. All the testing process is done automatically.

1 Introduction

The Focal environment [9], developed by the Focal project[1] (initiated by T. Hardin and R. Rioboo and further developed by researchers coming from laboratories LIP6, CÉDRIC and INRIA), allows one to incrementally build library components and to formally prove their correctness. A component of a Focal library can contain specifications, implementations of operations and proofs that the implementations satisfy their specifications. In the early development stages, components contain only specifications, then step by step components are refined and completed with implementations by a refinement mechanism based on inheritance. Proofs may be done at any time. The Focal environment incorporates a prover called Zenon [4] which can automatically discharge proof obligations, with the help of intermediate lemmas given by the user. Focal components are translated into OCaml executable code and are verified by the Coq proof assistant [12].

Even if the Focal environment ensures a high level of confidence because of its methodology based on specification and proof, we cannot do without testing. Here are some reasons:

– The user, based on the informal specification from the user or the domain expert, writes the formal system specification. In Focal, it consists in formal properties maybe distributed in different components. For some of them the developer will provide a proof that the code is correct with respect to this

[1] `http://focal.inria.fr`

formal specification. But some of the properties may not be proven, for example low level properties about the addition of machine integers (we trust them because of external formal formalizations) or very general mathematical properties. In the latter case, these properties are assumed to be true (the keyword `assumed` is used instead of giving a sketch of proof). These properties may become test objectives.

– In the context of a functional validation process, when it is independent from the validation done by the development team, engineers often verify by testing the correctness of the final software with respect to their own specification. Let us call this specification the external one. So thanks to the inheritance mechanism of Focal, these external properties can be encoded in Focal in a component, from which the implementation will inherit. Then as previously it becomes possible to verify by testing if the code satisfies these new properties.
– There exist some basic types in Focal, e.g. `int`. This type is translated into the Ocaml type `int` and the Coq type `Z`. So in the executable code, machine integers are used but proofs are done with inductively defined integers. So we have *some* confidence in the code but we must test code to verify if the properties are verified, in particular around the bounds.
– Some OCaml untrusted code may be imported in a Focal certified code. No proof is done on this imported code.
– When Zenon, the prover integrated with Focal, does not succeed in proving a property automatically, two issues are possible: either it is not true or Zenon needs to be helped by giving some intermediate lemmas the user has to find. So before beginning the latter expensive task, we can test a not yet proven property in order to discover a counter-example or to have more confidence in the property. It can also be used to have confidence in the lemmas we need to introduce while proving a property (e.g. invariants, technical intermediate lemmas, supplementary assertions). In this context, testing is used for debugging specifications and programs before a proof is attempted or while it is being attempted. Such testing facilities have been integrated into the Isabelle [2] or Agda/Alfa [10] proof assistants.

In this paper we propose to test the code with respect to the expected properties written by the specifier or expressly introduced by the tester as for instance metamorphic relations (as introduced by Chen, Tse and Zhou [6]). We describe the testing framework and the corresponding tool FocalTest. More precisely a property, considered as an executable predicate, is exercised with some randomly generated inputs. Experience shows this style of testing is a useful method for debugging programs and specifications as exemplified by the tool Quickcheck developed for Haskell by Claessen and Hugues [8].

The tool FocalTest automatically produces the test environment and the drivers to conduct the tests. We benefit from the inheritance mechanism to isolate this testing code, called the testing harness in the paper, from the components written by the programmer.

The paper is organized as follows. First we briefly present the environment Focal and its large-spectrum language also called Focal. Then in Section 3, we define the syntax of the properties allowed for testing and overviews the testing procedure. The generation of the testing harness is detailed in Section 4. We illustrate our purpose with the triangle example in Section 5. Section 6 proposes a coverage analysis. Lastly we mention some related work before some concluding remarks and perspectives.

2 The Focal environment and its language

The program development environment Focal is a framework dedicated to the complete development of certified components —in the sense of piece of specification/code proved correct with respect to the specifications— from the specification stage to the implementation one. In this section we give a brief overview of the underlying language, also called Focal. For further explanations please consult the documentation at `http://focal.inria.fr` and [9].

The language Focal is a functional language whose syntax is close to OCaml. It also incorporates some object oriented features such as inheritance, abstraction, late binding and redefinition. It allows us to define two kinds of structures, *species* and *collections*.

Roughly speaking a species defines a set of values that can be manipulated through functions called *methods*. At early stages in the development, those values and methods are abstract. For methods it means the user only writes their type, i.e. the types of the parameters and the result. He/she can also write specifications as properties involving the methods. As an example let us consider the species `Setoid` that specifies the notion of a set equipped with an equivalence relation `equal`:

```
species setoid  =
  rep;
  sig equal in self -> self -> bool;
  property equal_refl: all x in self, equal(x,x)
  property equal_sym: all x, y in self, equal(x,y) -> equal(y,x);
  property equal_trans: all x, y, z in self,
    equal(x,y) -> equal(y,z) -> equal(x,z);
end
```

This small example deserves some explanations about syntax: `self` is put for the type of the elements defined in the current species. The keyword `rep` introduces the type of the elements manipulated by the methods of the species. In the early development phases, it is usually abstract as in the example, it is later refined and defined as a concrete type *à la ML* called the *carrier type*.

Let us complete this species with a binary method `different` which returns `true` if its arguments are different and `false` otherwise. We can define this function although `equal` is not already defined (`#not_b` is the predefined operation on booleans) thanks to the mechanism of *late binding*. Furthermore we can demonstrate a property, that is `different` and `equal` are dual from each other. The proof is not detailed here —no proof will be shown in the paper— because its form does no matter in this paper.

```
let different(x,y)= #not_b(equal(x,y));
theorem different_not_equal: all x, y in self,
   different(x,y) <-> (not equal(x,y))
   proof:
   ...
```

Species may be defined from scratch but they are usually defined by using inheritance, more precisely multiple inheritance. Thus a Focal development forms a hierarchy whose nodes are species. Nodes close to a root correspond to pieces of specifications whereas deep nodes are made more and more precise, and then are close to implementations. Along inheritance paths, methods, carriers, properties can be refined (defined or redefined, proved in the case of properties). When a carrier type is defined in an inherited species, it cannot be redefined.

A species is said *complete* when every declared method (inherited or not) is defined and every stated property (inherited or not) has been proved or admitted (in such a case the proof is replaced by the special keyword `assumed`).

Collections are the implementations of species. A collection derives from a complete species. Collections are the leaves of the inheritance graph, cannot be refined by inheritance (like a *final* Java class for example). A collection is close to an abstract data type: it defines a type whose representation is abstracted and elements of the collection can only be manipulated with the help of the collection (those of the generating species, inherited or not).

The type of a collection is its interface obtained from the complete species the collection derives from: by removing definitions and proofs and abstracting the `rep` type in all the method types. The interface of a collection is named as the complete generative species it comes from. Interfaces can be ordered by inclusion, which gives a very simple notion of sub-typing.

Species can be parameterized by collections. The formal parameter is introduced by a name c and an interface I. Any collection CC having an interface including I can be used as an actual parameter for c. In the species body, the methods and properties of the parameter are denoted by `c!m`. The fact that CC has been created upon a complete species ensures that no link error can arrive at runtime and that proofs of CC can be used as lemmas. Species can also be parameterized by elements of collections, themselves introduced as parameters, thus introducing a dependence between parameters. Type-checking forbids dependence cycles between parameters.

3 Testing properties

3.1 Overview

Usually, software testing requires the definition of an oracle that will determine whether or not an input/output pair satisfies a given predicate. The oracle is traditionally the tester itself, another existing program or an executable specification. In this case, during the execution of a test case, the tester or the testing tool will compare the actual output with the expected output computed by the oracle in order to establish the verdict. Our motivation is to verify the code by testing some properties extracted from the specifications or expressly written from test purpose. Since a property defines an executable predicate, we just need to know if the target property holds or not for some valuations of its bounded variables. Thus properties serve as oracles in their general acception.

The only information required in the test of a property are the test set and the verdict of the calculus. We can consider the property under test as a tuple composed of some *pre-conditions* and a *conclusion* that will help us to decide if test data are relevant or not and to compute the test verdict.

Testing a property of a species S requires to execute the methods involved in the statement. Thus those methods need to be defined in S or inherited. Furthermore the carrier type must be defined at this stage in order to be able to design test cases. For simplicity we impose that S is a complete species (no matter whether the proofs are done or not, we do not care about them). This hypothesis can be relaxed without any difficulty. In fact, a dependency analysis, already implemented in the Focal compiler, is enough to verify if the property to be tested can be executed.

The property under test is either defined in the species or inherited. Thus it can have been written at any development stage and can be a very abstract one.

3.2 Testable Properties

Focal allows us to express a large class of properties. Because efficiently testing any property is not possible at first glance[2], we restrict ourselves to the class of *testable* properties which take the following form:

$$\forall X_1 : \tau_1 \ldots X_n : \tau_n . \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow (A_1^1 \vee \ldots \vee A_{n_1}^1) \wedge \ldots \wedge (A_1^m \vee \ldots \vee A_{n_m}^m)$$

where the α_i are produced by the grammar

$$\alpha ::= \alpha \vee \alpha | \alpha \wedge \alpha | A$$

The atomic formulas A and A_i^j are calls to Focal boolean methods, with an optional negation, and $\tau_1 \ldots \tau_n$ denote Focal types.So, testable properties are some first order formulas in prenex form without any existential quantifier. These formulas may contain free variables, the Focal compiler ensures that these variables are well defined somewhere in the species or the inheritance path.

We distinguish two parts in these properties: the pre-condition and the conclusion.

Definition 1. *Let $P \equiv \forall X_1 : \tau_1 \ldots X_n : \tau_n . \alpha_1 \Rightarrow \ldots \alpha_n \Rightarrow \beta$. We call the* pre-condition *(resp. the* conclusion*) of the property P, the predicate $Pre(P) = \alpha_1 \wedge \ldots \wedge \alpha_n$ (resp. $Con(P) = \beta$).*

[2] The $\exists$ quantifier is known to be a difficult problem.

3.3 Elementary properties

In order to test a property, we first transform it into a set of simpler properties called *elementary properties* by applying the rewriting rules detailed in Figure 1. All the properties issued from the transformation will be tested separately. They are all together logically equivalent to the initial property (see Theorem 1). The reason why we transform a property into a set of elementary ones is that the property may specify a large variety of behaviors. Intuitively, an elementary property specifies a more restricted effect.

$$\alpha_1 \Rightarrow \ldots \Rightarrow (\beta_1 \vee \ldots \vee \beta_m) \Rightarrow \ldots \Rightarrow \alpha_n \longmapsto \begin{cases} \alpha_1 \Rightarrow \ldots \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \alpha_n \\ \alpha_1 \Rightarrow \ldots \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \alpha_n \\ \vdots \\ \alpha_1 \Rightarrow \ldots \Rightarrow \beta_m \Rightarrow \ldots \Rightarrow \alpha_n \end{cases}$$

$$\alpha_1 \Rightarrow \ldots \Rightarrow (\beta_1 \wedge \ldots \wedge \beta_m) \Rightarrow \ldots \Rightarrow \alpha_n \longmapsto \alpha_1 \Rightarrow \ldots \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_m \Rightarrow \ldots \Rightarrow \alpha_n$$

$$\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow (\beta_1 \wedge \ldots \wedge \beta_m) \longmapsto \begin{cases} \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta_1 \\ \vdots \\ \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta_m \end{cases}$$

Fig. 1. Rewriting system

In the rewriting rules (Figure 1), quantifiers are omitted. The first rule consists in eliminating a disjunction appearing in the left hand side of a property, it creates a set (more precisely a multi-set) of properties. Intuitively, it corresponds to a case analysis. The second rule transforms a conjunction in the left hand side by its equivalent form with implications. The third rule splits the conjunction in the right hand side of the last implication. Like the first rule, it creates as many properties as sub-formulas in the initial right hand side conjunction.

These transformation rules constitute a rewriting system. It terminates (trivial by considering the number of $\Rightarrow$ and $\Leftrightarrow$ occurrences and the number of $\vee$ and $\wedge$ occurrences) and is confluent (all critical pairs can be joined). So every testable property P can be rewritten in a normal form (each formula of the set obtained from a rewriting step is again rewritten until convergence), which is a multi-set of formulas written $P^*_\downarrow$. The elements of $P^*_\downarrow$ are called the *elementary properties* of the original property. They have the following form:

$$\forall X_1 : \tau_1 \ldots X_n : \tau_n.\ A_1 \Rightarrow \ldots A_n \Rightarrow B_1 \vee \ldots \vee B_m$$

where A_i and B_i are atomic formulas.

Theorem 1. *Let P the property $\forall X_1 : \tau_1 \ldots X_n : \tau_n.A_1 \Rightarrow \ldots A_n \Rightarrow B_1 \vee \ldots \vee B_m$. So P is equivalent to* $\displaystyle\bigwedge_{f \in P^*_\downarrow} \forall X_1 : \tau_1 \ldots X_n : \tau_n.f$

3.4 Test procedure

The original property is not considered in the test procedure. It is replaced in this process by its elementary properties. Each elementary property is considered and tested separately. Thus each elementary property has its own test set (composed of independent test cases).

A test case is a valuation σ which maps each quantified variable X_i to a value. It is randomly generated; we detail the generation in a next section. The elementary property $\forall X_1 : \tau_1 \ldots X_n : \tau_n.A_1 \Rightarrow \ldots A_n \Rightarrow B_1 \vee \ldots \vee B_m$ is then checked by considering its pre-condition and its conclusion in two steps:

- firstly, the pre-condition is evaluated with respect to σ. This is the validation part of the test case. If the pre-condition reduces to *false* or *fails*, the test case is rejected as being irrelevant. If it evaluates to *true*, go on with the next step;
- lastly, if the test case passes the pre-condition, we can compute the verdict. For that purpose, we evaluate the conclusion with respect to σ. If the result is *true*, then the verdict is OK. If it is *false*, the verdict is KO and we have found a counter-example that exemplifies the property is not satisfied for that test set. Anf if an exception is raised, the tester should decide himself if the exception is expected or not.

4 Test harness

In this section we describe the test environment and the drivers we automatically produce to conduct the tests.

4.1 Structure

Our tool does not modify the species S that contains the property to be tested, FocalTest automatically derives a species $SHarness$ from S, called here the *test harness* of S. This species principally contains a method `random` of type `int -> self` which generates random values of the carrier type, a method `test_prop` which implements the test loop and a method `gen_report` which produces the testing report (e.g. in XML format).

For the synthesis and execution of test cases, we need to create and manipulate some data of the types given for the quantified variables of the property under test.

The type of a quantified variable in a property can be `self`, a basic Focal type `int`, `bool` ..., a concrete ML like type, a cartesian product or one of the abstract types described by the collections which may parameterize the species under test. In the latter case the type receives the name of the parameter. For example, in a species S parameterized by a collection C we can use the type C in particular to describe the carrier type of S (e.g. `rep = int * C` means an element of the species S is represented by a pair composed of an integer and an element of C).

We suppose the methods which generate values for the basic Focal types are known. In the case of the `self` type, we need the associated concrete representation. It is available since we have assumed the species is complete. So, FocalTest will produce the data generator by following the structure of the type (see next section for more details). In the case where S is parameterized by a collection C of interface $S1$ and when the carrier type refers to a parameter of the species (e.g. `rep = int * C`), the generator of *rep* values will call the generator for values of type C. So, in this case the harness of S is a species parameterized by a collection C' whose interface is $S1Harness$ that is the harness derived from $S1$.

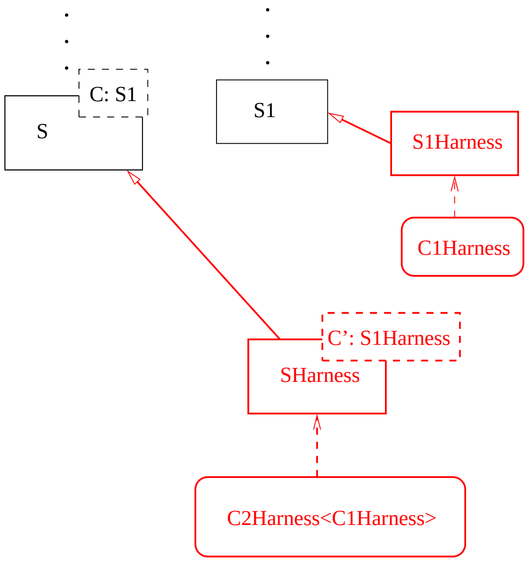

Fig. 2. The test hierarchy: target and harness

By extension, we call harness the set of species which add the random generators and the testing loop.

Figure 2 shows an example of a Focal hierarchy equipped with harness. Square boxes represent species (complete species in our context) whereas rounded boxes represent collections. Dotted arrows represent abstraction links between a collection and the species it is built from (e.g. `C1` and `S1`). Plain arrows represent inheritance dependencies (e.g. `S2Harness` inherits from `S2`). The parameter of a species is represented by a small dotted rectangle in the right upper corner (e.g. `S2` is parameterized by `C` of interface `S1`). When an instance is created, the effective parameter is indicated between `<` and `>` (e.g. the collection `C2` is the result of the application of `S2` on the parameter `C1`). In this example `S1` is complete. Finally FocalTest creates the collection `C2Harness` by applying `SHarness` to `C1Harness`, a collection built from `S1Harness`.

4.2 Test data generation

The FocalTest tool automatically creates the methods which pseudo-randomly generate values for the quantified variables of the test target. For each type τ appearing in the target property, FocalTest automatically defines a generator which can produce values of this type. its body is created by following the structure of τ. For a product type $\tau_1 * \tau_2$, it means FocalTest generates firstly the method for the type τ_1, secondly the method for the type τ_2 and lastly the methods for the product by combining the two previous generators. Focal allows to define concrete types that are defined by enumerating the values constructors. In the case of a concrete type, FocalTest generates for each constructor the generators for the constructor parameters and then combines them into the method generating values for the full type. In the case of the special imported OCaml types like *int*, FocalTest relies on the existing methods and imports them. In case of recursive data-types, we first choose the nature of the constructor, recursive/non-recursive with probability 1/2. Then we take uniformly a constructor in the chosen family.

Our approach to generate random values is a naive one. The distribution is not uniform and the generators tend to generate small sized values. However, they do not exclude any value, in other words, the functions are surjective. This can be improved, taking benefit from work as for instance [11].

5 FocalTest Experimentation

This section illustrates the usage of FocalTest on a classical example in testing literature, the triangle type program. The program takes the three lengths of the sides of a triangle and returns the nature of the triangle formed by input lengths: `Equilateral`, `Isosceles`, `Scalene` or `Error` if the three lengths do not define a triangle. So, the output of the program is the next Focal type:

```
type triangle_type =
  Equilateral in triangle_type;  Isosceles   in triangle_type;
  Scalene     in triangle_type;  Error       in triangle_type;
```

The length of a triangle edge is represented by an integer considered as an element from a commutative monoid. Thus, triangles are entities of a collection whose carrier type is a 3-tuple of lengths. The method implementing the specification given upper is named `type_triangle`, it has type `self` $\rightarrow$ `triangle_type`.

Two kinds of properties, soundness and correctness properties, are defined to specify the link between the arguments and the returned value of `type_triangle`. The soundness properties specify which constraints on lengths hold when the method `type_triangle` returns a specific value. The completeness properties specify which value can be returned by `triangle_type`.

The properties are shown in Figure 3. We have only detailed some of them because lack of space. The property `triangle_type_correct_equiv` states that if the method `triangle_type` returns the value `Equilateral` for the triangle

```
property triangle_type_complete: all t in self,
  triangle_type(t) = Equilateral or triangle_type(t) = Isosceles or
  triangle_type(t) = Scalene or triangle_type(t) = Error;

property triangle_type_correct_equiv: all t in self,
  triangle_type(t) = Equilateral ->
  (edge!equal(fst(t), snd(t)) and edge!equal(fst(t), thrd(t)) and
  edge!equal(snd (t), thrd(t)) and edge!gt(fst(t), edge!zero))
```

Fig. 3. Some properties about *type_triangle*

`t`, then its three lengths are equal and greater than zero. The other correctness properties are similar.

This Focal development has been tested under FocalTest. The integers implementing lengths were constrained to be chosen in the interval 0–10. We tested 14 properties among those that were specified. These ones led to 40 elementary properties and asked FocalTest to generate 10 test cases for each property. This experiment detected no bugs. The test generation and execution were immediate. A potential overhead can be observed because of the harness compilation. For a large majority of the properties, less than 100 irrelevant test cases were required before obtaining the 10 valid required test cases. Some properties asked for about 1 000 irrelevant test cases.

For evaluating the quality of our testing tool, we created 10 mutants of the triangle program. We used mutation operations such as the replacement of an operator or a connector by another one (e.g. $\leq$ by $\geq$, $\wedge$ by $\vee$), the replacement of a variable by another one in a property, the replacement of a constant by another constant. We have evaluated the capacity of FocalTest to kill mutants. For this purpose, FocalTest has been run on each mutant several times, each time with new randomly generated test data, with the same parameters as previously. We can notice three behaviours among the 10 mutants. 2 mutants led to properties with unsatisfiable preconditions, a timeout was raised after 100 000 invalid test cases for each execution of FocalTest. Another mutant was never killed, indeed the domain (1–10) we chose was too restrictive and negative values should have killed the mutant (an experimentation with such a domain for lengths allowed us to confirm it). The 7 remaining mutants were killed every time FocalTest was run.

6 Coverage analysis

Before defining some coverage criteria, we formalize the notion of pre domain and establish the basis of our testing method.

6.1 Pre-conditions and Pre-domains

The pre-condition and the conclusion of a testable property play a fundamental role. Intuitively, the pre-condition defines a set of values.

Theorem 2. *Let P_1 and P_2 such that $P_1 \longmapsto P_2$, then $Pre(P_2)$ implies $Pre(P_1)$.*

Proof. *All rules but the first leaves the pre-condition unchanged. So we have to prove the fact for the first rule only. In that case, the pre-condition changes from $\alpha_1 \wedge \ldots \wedge (\beta_1 \vee \ldots \vee \beta_m) \wedge \ldots \wedge \alpha_n$ to the pre-conditions $\alpha_1 \wedge \ldots \wedge \beta_i \wedge \ldots \wedge \alpha_n$ for some $i \in [1, m]$. The conclusion is then obvious.*

The next theorem allows us to extend the previous property to an elementary property of P.

Theorem 3. *Let P by a testable property and P' an elementary form of P. Then $Pre(P')$ implies $Pre(P)$.*

So any elementary property of a testable property P has a pre-condition weaker than the pre-condition of P. So any valid test case for an elementary property of P is a valid test case for P.

Definition 2. *Let $P \equiv \forall X_1 : \tau_1 \ldots X_n : \tau_n . \alpha_1 \Rightarrow \ldots \alpha_n \Rightarrow \beta$ be a testable property. We call* pre-domain of P, *the set $PrD(P)$ where $(v_1, \ldots, v_n) \in PrD(P)$ if and only if $Pre(P)$ holds for $X_1 = v_1, \ldots, X_n = v_n$.*

Intuitively, for a property P, $PrD(P)$ defines the set of all valuations σ which validate the pre-condition of P. The following theorem shows us the link between a property and its elementary forms according to the notion of pre-domain.

Theorem 4. *If a property P' is an elementary form of a property P then $PrD(P') \subseteq PrD(P)$*

Proof. *Since, $Pre(P')$ implies $Pre(P)$, $PrD(P') \subseteq PrD(P)$ follows.*

Hence, the pre-condition of each elementary form can be considered as the definition of a domain, identifying a kind of equivalence class of the pre-domain of the initial property. Because the pre-domain of an elementary form is a subset of the pre-domain of the original property, we can consider an elementary property as a sub-property. The original property is the combination of these sub-properties. So testing these properties separately is a gain since we have a finer granularity.

By the last theorem, all elementary forms of a property define a domain of test cases which is a subset of the original property's domain. But we should prove we do not loose any element of the pre-domain of P by considering only the elementary properties. Any test case in the pre-domain of P should be in the pre-domain of, at least, one elementary property of P.

Theorem 5. *Let P be a property. Let P'_1, ..., P'_n the properties resulting from the application of a rewriting rule on P. Then, $PrD(P) = \cup_{i=1}^{n} PrD(P'_i)$.*

Proof. *All rules but the first one leave the pre-condition unchanged. So the property is immediately true for these rules. For the first rule, if $Pre(P) = \alpha_1 \wedge \ldots \wedge (\beta_1 \vee \ldots \vee \beta_m) \wedge \ldots \wedge \alpha_n$ then $Pre(P'_i) = \alpha_1 \wedge \ldots \wedge \beta_i \wedge \ldots \wedge \alpha_n$. So, $\cup_{i=1}^{n} PrD(P'_i) = \{v_1, \ldots v_m | (v_1, \ldots, v_m) \in \cup_{i=1}^{n} PrD(P'_i)\}$. Also, by definition of PrD, $(v_1, \ldots, v_m) \in \cup_{i=1}^{n} PrD(P'_i) \leftrightarrow Pre(P'_1) \vee \ldots \vee Pre(P'_n)$ holds* *for* $X_1 = v_1, \ldots, X_m = v_m$*. We prove by definition of* Pre *that* $Pre(P'_1) \vee \ldots \vee Pre(P'_n) \Leftrightarrow Pre(P)$*. And so* $\cup_{i=1}^{n} PrD(P'_i) = \{v_1, \ldots, v_m | (v_1, \ldots, v_m) \in PrD(P)\} = PrD(P)$*.*

Theorem 6. *Let* P *be a testable property. Then* $PrD(P) = \bigcup_{P' \in P^*_\downarrow} PrD(P')$*.*

The last theorem (following from Theorem 5 and associativity of $\cup$) tells us that the rewriting system preserves pre-domains. Testing the elementary properties separately is complete; any test case relevant for the original property is a possibly test case for at least one elementary property. Two pre-domains may overlap or even be equal (for example the third rule creates many properties all sharing the same pre-domain). It would be interesting to detect that two non equal pre-domains overlap. It probably means that the original property contains some redundant parts.

An elementary form coverage criteria consists in considering all the elementary properties obtained by the rewriting rules except the third one (to avoid pre-condition duplication). Then for each elementary property P'_1, select a test case in $PrD(P'_1)$ which is not a member of $PrD(P'_2)$ for some other elementary form P'_2. When a test case belonging to the pre-domains of two different elementary properties is discovered, it is worth reporting it.

6.2 A MC/DC like criteria

In the last section, we have proposed a first coverage criteria. Since a pre-condition can be considered as a decision, we explore some decision coverage. More precisely we are interested in the MC/DC coverage.

In the MC/DC criteria we have to demonstrate that every condition in a decision changes the outcome of the decision independently of the other conditions. For this purpose, for each condition there should be two test cases where the condition evaluates differently while the other conditions evaluate to the same value while the outcome of the decision is modified for both test cases. In a property (or an elementary form), the pre-condition and the conclusion are both decisions. A MC/DC style criteria for a set of elementary forms consist in applying for each elementary form the following scenario:

– select a test set satisfying the MD/DC criteria on the pre-condition. Because the pre-condition is a conjunction of a number of conditions, only one test set can be applied. It requires one test case where all decisions are evaluated to *false* (so the outcome of the pre-condition is *false* also). And for each decision, one test case where this decision evaluates to *false* while the other ones evaluate to true. It requires $n + 1$ test cases where n is the number of decisions;
– select a test set satisfying the MC/DC criteria on the conclusion: i) a set of test cases where all conditions but one evaluate to *false*. Each test case

should evaluate the pre-condition to true; ii) a test set where all conditions evaluate to *false*.

For the first requirement, we ensure that the pre-condition can be evaluated to *false*. If the pre-condition cannot be evaluated to *false*, it means the pre-condition plays no role in the elementary form. This would emphasize that in the property (before rewriting) there is a part of the pre-condition without any effect. It also ensures each element of the pre-condition has an effect. A condition of the pre-condition which cannot be set to *false* means the pre-condition of the original property contains some useless parts.

With the second requirement, we ensure first, in the case the pre-condition is true, all conditions in the conclusion are independent from each other and they can all together set the conclusion to true. Like for the pre-condition, if a condition of the conclusion is not independent from the others, it means the conclusion of the original property contains useless parts. The second requirement ensures also that the conclusion can be evaluated to *false*.

Presently FocalTest is able to calculate the coverage of such a criterion by the generated test sets, but only for the conclusion of each elementary property. Pre-conditions are not yet taken into account because as soon as a test case valuates a precondition to *false*, FocalTest rejects it. We have run FocalTest 10 times on the triangle example (10 test cases per property). We have obtained a rate of 76% in the coverage of the conclusions, as defined previously.

7 Related work

A lot of works have been done in the area of testing, especially for imperative languages and more recently for object oriented languages. For functional languages, the interest is more recent. One of the most advanced tools for testing functional programs is probably QuickCheck a tool for testing Haskell programs [8]. It provides a powerful specification language based on the first order logic and offers some combinators to write specification. The user has to type a correctness property and sends it to QuickCheck. The property could be a simple predicate or a more complex one with a pre-condition part. The tool analyses the type of the proposition and generates randomly test data, submits them and calculates the verdict for an arbitrary number of test cases. For a more effective use of QuickCheck, the user may define his own data generator. For example, if a property deals with a sorted list of integers, the user can provide a generator which only returns sorted lists. The Quickcheck approach and its good evaluation by users have inspired our own approach. Gast [13] is similar to Quickcheck for the language Clean. The user does not have to supply test data generators, they are automatically generated for arbitrary data types. On this point, we share the same particularity. But for recursive types, Gast does not randomly select the size of the values, it performs a breadth-first enumeration. And so, it is usually limited to small sized values, e.g. lists. On the contrary FocalTest tries to generate random values by distinguishing recursive constructors and non-recursive constructors and chooses one of them with a uniform distribution.

Other tools integrated in proof assistants have been inspired by the previous approach. For example, in Isabelle [2] by Berghofer and Nipkow or in Agda [10] by Dybjer, Haiyan and Takeyama. They allow the user to test some theorems before attempting a proof. and thus to debug specifications and proofs.

Another initiative has been proposed and implemented in Isabelle/HOL. The testing tool HOL-Testgen [5] on top of Isabelle-HOL aims to add some unit testing features. It allows the user to write test specifications. The tool partitions the input space of the specification and generates automatically the test script in SML. The implementation is then tested. HOL-Testgen exploits the common testing hypothesis formalized in [3], e.g. the regularity hypothesis. A regularity level k hypothesis means that if an implementation satisfies the requirements for test data of size less or equal than k then the implementation is correct for all data.

Our approach considers a formal specification as a test oracle. A decision procedure for the test oracle is automatically derived from the specification. Many researchers have proposed such an approach, e.g. [1, 7]. We do not use a runtime assertion checker directly on the assertions written by the user but on an equivalent set of more tractable and traceable properties (for coverage computations for example).

8 Conclusion and future work

In this paper we have presented the FocalTest tool that permits to validate one or several components with respect to the specifications written in them. It can be used *a posteriori* or during the development process to debug specifications and implementations or also to have some confidence in a property before proving it. Although the case study presented in the paper is a small one, it demonstrates that our approach and its associated tool, FocalTest, are useful to find bugs. Furthermore, FocalTest has been used on the Focal standard library itself. It has permitted to reveal an error in a component: a comparison operator was wrong in a property which was not proven. In that case, the code was correctly written but the specification was not.

We rely on randomly selected test cases. A first requirement is put on these test cases: they must satisfy the pre-condition of the property under test. We can repeat the random draw until convenient values are produced but it can be an expensive process for some kind of pre-condition. To overcome this drawback, several solutions can be proposed. A first one is to provide the user with the possibility to define a specific purpose data generator tuned to generate valid test cases. Another method consists in exploring very carefully the pre-condition and more precisely the definition of the involved methods in order to produce constraints upon the values of the variables. Then it would remain to instantiate the constraints in order to generate test cases ready to be submitted. This method is a *white box* method testing whereas the currently implemented method is a *black box* testing method. This direction is one of our perspectives to improve our testing method and is currently under study.

References

1. S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. Softw. Eng.*, 26(1):55–69, 2000.
2. S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.
3. G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6), 1991.
4. R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *LNCS*, pages 151–165. Springer, 2007.
5. A. D. Brucker and B. Wolff. Test-sequence generation with hol-testgen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007: Tests And Proofs*, volume 4454 of *LNCS*. Springer-Verlag, Zurich, 2007.
6. T. Y. Chen, T. H. Tse, and Z. Zhou. Fault-based testing without the need of oracles. *Information & Software Technology*, 45(1):1–9, 2003.
7. Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, volume 2374, pages 231–255, London, UK, 2002. Springer-Verlag.
8. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
9. C. Dubois, T. Hardin, and V. Viguié Donzeau-Gouge. Building certified components within focal. In H.-W. Loidl, editor, *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004, München, Germany*, volume 5 of *Trends in Functional Programming*, pages 33–48. Intellect, 2006.
10. P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In D. Basin and B. Wolff, editors, *Proceedings of Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 188–203. Springer-Verlag, 2003.
11. P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. 2007. Chapters I-IX, Draft available electronically from P. Flajolet's home page.
12. INRIA. Coq*, version 8.1*, Nov. 2006. Available at: `http://coq.inria.fr/`.
13. P. W. M. Koopman, A. Alimarine, J. Tretmans, and M. J. Plasmeijer. Gast: Generic automated software testing. In R. Pena and T. Arts, editors, *Implementation of Functional Languages, 14th International Workshop, Madrid, Spain, September 2002, Revised Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer, 2003.