UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Testing and debugging filtering of global constraints

Dissertation presented by
Aurélie MASSART , Valentin ROMBOUTS

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Pierre SCHAUS

Reader(s)
Pierre SCHAUS, Yves DEVILLE , Sascha VAN CAUWELAERT

Academic year 2017-2018

Abstract

Today, Constraint Programming becomes more and more useful to solve complex problems in many fields. Many problems could be modelled using Constraint Programming, including real life problems such as for example organizing train schedules minimizing delays. The modelling of such problems requires the creation of constraints. Each constraint allows to reduce the number of solutions by filtering the domains thanks to filtering algorithms. Many Constraint Programming solvers use filtering algorithms to quickly find solutions to problems. Nevertheless, those filtering algorithms need to be tested, because if they contain bugs, the solutions would not necessarily be correct. Nowadays, solvers are mostly tested by writing test-suites without any supporting tool specific to Constraint Programming, taking a lot of time for developers. This is the reason why creating a library to ease the testing of filtering algorithms is interesting. This master thesis presents a library able to debug the domains' filtering procedure of any constraint acting over integers in any language based on the *Java Virtual Machine* such as *Java* or *Scala*.

Acknowledgements

We would like to sincerely thank our advisor Pierre Schaus for his support during the year. He shouldered and helped us during the hard times and proposed some solutions to guide us towards the realization of this master thesis.

We are also particularly grateful to Sascha Van Cauwelaert for answering some of our questions and coming with innovative ideas to realize incremental tests which became an essential part of this master thesis.

Finally, we want to thank our relatives and friends for their support during the realization of this master thesis.

Contents

Chapter 1

Introduction

Nowadays, Constraint Programming becomes a popular way of solving problems. Indeed, a lot of problems can be modeled and solved thanks to Constraint Programming. This includes simple problems such as solving a sudoku or complex problems such as organizing the train schedules to minimize delays. To solve them, many Constraint Programming algorithms are constantly created. Some tools called CPSolvers are developed to solve Constraint Programming problems by modeling them with variables and constraints before applying algorithms to find solutions to the problems. But how to check that these implementations work? How to test them? This thesis is about a tool able to test such algorithms.

More specifically, these Constraint Programming algorithms consider variables having domains and should filter the domains according to some constraints. By filtering the domains, we mean removing some values that are not part of a solution to the problem posed (do not satisfy some constraints). The tool we created, let's call it "*CPChecker*", tests the filtering algorithms related to a constraint.

CPChecker is intended to help the developers to debug their filtering implementations. This tool should be generic in order to be able to test the filtering of any constraint acting over integer values. It can be very useful to dispose of such a tool for Constraint Programming developers. Indeed, making tests for each constraint's filterings would take a lot of time. Using that type of tool instead would allow the developers to save time.

Today, Constraint Programming solvers' filtering implementations are mainly tested using test suites. For example, the ***Choco***[1] solver contains unit tests created using the testing framework *TestNG*[1]. The ***JaCoP***[2] solver is exhaustively tested with frequent Constraint Programming problems. These unit tests are created using the *JUnit*[2] testing framework. For the ***OscaR***[3] solver, which is written in *Scala*, the filtering algorithms are also unit tested, sometimes by comparing the number of solutions found by different algorithms over some Constraint Programming problems. It uses *ScalaTest*[3], a unit test framework for *Scala* code. Creating such unit tests is a good way for testing the filtering algorithms, however writing all these lines of code can be very time consuming. It is the reason why using a tool automating tests for the filtering algorithms could be a good asset for solver developers.

At this stage, one can wonder how we can be sure that the tool works correctly. How to trust our implementation? The most simple way to reach this objective lay in the readability of the tool. *CPChecker*'s algorithms should be made simple and readable so that the user should be

[1] http://testng.org/doc/
[2] https://junit.org/junit5/
[3] http://www.scalatest.org/

able to easily understand the implementation of the tool simply by reading it. As it is often said: "understanding is believing".

This thesis is separated in 4 main parts. Firstly, the concepts of Constraint Programming used in this thesis will be summarized in the *Constraint Programming* chapter. The basic concepts will be explained and some constraints will be observed in more details. This chapter will then be concluded with concrete examples of Constraint Programming models.
Secondly, we will present the tool created in this thesis. In this chapter, the different ways to use it will be explained and illustrated over examples. Its implementation at key points of the code will also be explained.
In the third chapter, the performances of *CPChecker* will be analyzed, while possible extensions will be proposed in chapter 4, before concluding. Before going into the heart of this thesis, we will present briefly the state of the art in this subject.

1.1 State of the art

The notion of Constraint Programming already existed decades ago. A huge number of articles have been written about this subject but only a few of them talk about debugging tools for this domain. Here is a short overview of what has already been done over Constraint Programming debugging.

In 1995, the authors of [4] created a debugging tool on top of ECL^iPS^e. This tool allows to follow the evolution of the variables' domains during the execution of the algorithm through a graphical interface. But since this tool is integrated into ECL^iPS^e, it is not generic and cannot be used to test several CPSolver's filtering algorithms.

One year later, an interactive visualization tool for the *OZ* language has been created [5]. This tool displays the relevant information about the constraints through a search tree. Once again, this tool is *OZ* developers oriented and cannot be used to test solvers that are not written in *OZ*.

In 2005, the authors of [6] have proposed a new way of visualizing constraint problems by displaying the constraint store. In this way, the user can have a structured hierarchy to represent the store of a Constraint Programming problem.

Compared to those articles centered about Constraint Logic Programming and languages such as *OZ*, *CPChecker* will have a new approach to debugging in the field of Constraint Programming. It will be able to be used in the *Java* and *Scala* languages. In contrast to some of those articles, *CPChecker* will not become a visualization tool but more like a testing tool. It will be used for correctness purposes. Therefore, it is closer to something like *JUnit* than a visualization tool.

As those articles, *CPChecker* will use the concepts of constraints and searches. But, it will focus on filtering algorithms. More precisely, on filtering algorithms for global constraints.

Chapter 2

Constraint Programming

This chapter will present the different principles and notions of Constraint Programming. It will regroup the different terminologies used in this thesis and explain them through examples.

2.1 Basic Concepts

The very first notion to know is what Constraint Programming refers to. Therefore, a short but simple definition of Constraint Programming has been written below.

Definition 1 *Constraint Programming refers to the modelling and resolution of problems using a set of constraints C_i defined over variables X_j.*

In this definition, two important notions have been used: 'constraint' and 'variables'. Knowing those two notions plus the notion of 'solver' allows everyone to globally understand the concepts of Constraint Programming. While the constraints and variables are used to model a problem, a solver is used to find solutions to this problem using the given constraints and variables. Let's first define what is a variable.

Definition 2 *In Constraint Programming, a variable represents an arbitrary value taken among a set of possible values called the domain of the variable. If the domain possesses a single possible value, the variable is said to be fixed to this possible value.*

With this definition, a variable can take anything as a value. Be it an integer, a double, a character, a name, etc. In this thesis, only the integers are considered. That is because those are the most used in Constraint Programming and many constraints are over variables with domains composed of integers.
A variable can be assigned to a possible value of its domain. A variable is assigned to a value v if its domain is reduced to only containing v. Given a set of variables, when all the variables of this set are assigned, this new set is called an instantiation of this set of variables.
With this vocabulary about the notion of variable, the notion of constraint can be defined.

Definition 3 *A constraint is defined over a set of variables. A constraint is a relation over this set.*

The relation between a constraint and its set of variables permits to state some definitions about a constraint.

Definition 4 *A constraint is said satisfiable if and only if there exists an instantiation of its variables respecting the relation it defines. Otherwise, the constraint is said to be unsatisfiable. A constraint is satisfied by an instantiation of the set of variables if and only if this instantiation respects the relation defined by the constraint. If the relation is not respected, the constraint is said to be unsatisfied.*

Now, the last important notion of Constraint Programming can be defined thanks to the definition of a constraint satisfaction.

Definition 5 *A solver is a tool that resolves a problem of Constraint Programming by finding instantiations of the variables satisfying all the constraints (called solutions).*

2.1.1 Different Types of Propagation

To solve a problem, a solver often tries to look at all the instantiations of a problem but only a few of those are solutions. Therefore, a solver filters some of those instantiations by propagating its constraints.

Definition 6 *Propagating/filtering a constraint refers to the reduction of the domains of the variables by taking off values which can not be part of an instantiation while satisfying the constraint.*

The algorithms used to propagate a constraint are called *filtering algorithms*. Those are the algorithms that are tested in this thesis. Most of the time, when propagating a constraint, solvers try to attain some properties over the constraints. The most commons are the arc consistency and the bound consistency.

Arc Consistency

The arc consistency (or AC) is a very strong property and if it was possible, a programmer would want to attain arc consistency for all the constraints. But filtering algorithms reaching arc consistency are too slow for a lot of different constraints. Here is the definition of arc consistency:

Definition 7 *The domains of the variables are said to be arc/domain consistent according to a constraint $\mathcal{C}$ if and only if for every possible assignment of each variable $\mathcal{X}_i$, the constraint is satisfiable.*

$$\forall i \ \forall x \in dom(\mathcal{X}_i) : \mathcal{X}_i = x \Rightarrow \mathcal{C} \ is \ satisfiable$$

In other terms, all values of each of the domains are part of an instantiation satisfying the constraint. To have a better understanding, let's take an example.
Let x, y, z be three variables. They all have the same domain: $\{1, 2, 3, 4\}$.
We define a constraint over those variables:

$$x + y < z$$

Now, let's reach arc consistency on this constraint by reducing the domains of x, y and z. First, we look at x and we notice that the constraint cannot be respected if $x = 3$ or $x = 4$. In fact, if $x = 3$, $3 + y$ is always greater than or equal to z. Idem for $x = 4$. Therefore, the domain of x can be reduced to $\{1, 2\}$. y is in the same position as x. Therefore, y can not be 3 or 4 while satisfying the constraint. The domain is thus reduced to $\{1, 2\}$ as for x. Now, if we look at $x + y$, we notice that $x + y >= 2$. z must be greater than 2 to satisfy the constraint. Then the domain of z can be reduced to $\{3, 4\}$.
Now the domains are arc consistent. Let's prove it. First, let's rewrite the domains :

$$dom(x) = dom(y) = \{1, 2\}$$

$$dom(z) = \{3, 4\}$$

To prove those domains are arc consistent, each value must be part of an instantiation satisfying the constraint. If x is set to 1, we can have $y = 2$ and $z = 4$ which satisfy the constraint. If x is set to 2, we can have $y = 1$ and $z = 4$. With those two instantiations, the only value not verified is $z = 3$. But this value satisfies the constraint when $x = y = 1$. Therefore, with the three instantiations $[(1, 2, 4), (2, 1, 4), (1, 1, 3)]$ all satisfying the constraint, the definition is satisfied and those domains are arc consistent according to this constraint.

Bound Consistency

There exists different types of bound consistency. All these types have in common that, to reach bound consistency, we only filter the bounds of each variable's domain. In [7], the authors propose definitions for different types of bound consistencies : the bound($\mathcal{D}$), the bound($\mathcal{Z}$) and the bound($\mathcal{R}$) consistencies. Here, we will only focus on the bound($\mathcal{D}$) and bound($\mathcal{Z}$) consistencies because the bound($\mathcal{R}$) one concerns real domains values. But in this thesis, we only consider integer domains' values.

Here is a reformulation of the definitions of bound($\mathcal{D}$) and bound($\mathcal{Z}$) consistencies according to [7] :

Definition 8 *The domains of the variables are said to be bound($\mathcal{D}$) consistent according to a constraint $\mathcal{C}$ iff for each variable $\mathcal{X}_i$, the following holds :*

$$\forall x \in \{min(dom(\mathcal{X}_i)), max(dom(\mathcal{X}_i))\} : \ \mathcal{X}_i = x \Rightarrow \mathcal{C} \ is \ satisfiable$$

This definition simply states that to be bound($\mathcal{D}$) consistent, the bounds of each variable must belong to a solution according to the constraint.

The definition of bound($\mathcal{Z}$) consistency requires a new notion : a domain interval. Here is a definition of a domain interval :

Definition 9 *The domain interval $\mathcal{I}$ formed by a domain $\mathcal{D}$ is defined by the smallest and biggest value (minimum and maximum) of the domain and refers to all the integer values between these minimum and maximum included:*

$$\mathcal{I}(\mathcal{D}) = [min(\mathcal{D}), max(\mathcal{D})]$$

For example, the domain interval formed by the domain $\{1, 3, 5\}$ is the set $\{1, 2, 3, 4, 5\}$.

With this notion of interval, the bound($\mathcal{Z}$) consistency can be defined :

Definition 10 *The domains of the variables are said to be bound($\mathcal{Z}$) consistent according to a constraint $\mathcal{C}$ iff for each variable $\mathcal{X}_i$, the following holds :*

$$\forall x \in \{min(dom(\mathcal{X}_i)), max(dom(\mathcal{X}_i))\} : \ \mathcal{X}_i = x, \ and \ \forall j \neq i, \ \mathcal{X}_j \in \mathcal{I}(dom(\mathcal{X}_j)) \\ \Rightarrow \mathcal{C} \ is \ satisfiable$$

The bound consistency has some similarities with the arc consistency. For both of them, some values must be part of a solution. For the arc consistency, it is all the values and for the bound consistency, it is about the minimum and maximum of every variables. As for the arc consistency, let's take an example, for the bound($\mathcal{Z}$) consistency.

Let x, y, z be three variables. They have the respective domains: $\{1, 6, 15\}$, $\{5, 6\}$ and $\{10, 13, 20\}$.
Let the constraint be $x + y + z = 20$.
We first look at x. More precisely, we look at its minimum : $x = 1$. In that case, having $y = 5$ and $z = 14$ satisfies the constraint. It is important to notice that 14 is not included in the domain of z even so z can be 14 because we look at its interval to find a solution. Note however that, if we have considered bound($\mathcal{D}$) consistency, this would not be considered as a valid solution, since 14 does not belong to the domain of z. After the minimum, we look at the maximum of x i.e. $x = 15$. It cannot satisfy the constraint because $15 + y + z >= 30$. Therefore, the domain of x should be reduced : $dom(x) = \{1, 6\}$. Second, we look at y but y cannot be reduced because its minimum and maximum are parts of a solution. For z, we notice that 20 cannot satisfy the

constraint. Therefore, $dom(z) = \{10, 13\}$.

Those domains are not yet bound consistent, so we look again at the minimums and maximums to see if the domains can be more reduced. Last time, the maximum of x was 15 but now it is 6. This value cannot be part of a solution therefore the domain of x is reduced to $\{1\}$. For y, last time the domain could not be reduced but notice that the interval of z is now only $\{10, 13\}$. With this domain, when $y = 5$, the constraint cannot be satisfied. After y comes z. z can also be reduced. Its minimum 10 cannot satisfy the constraint. Therefore, the domains become:

$$dom(x) = \{1\}$$
$$dom(y) = \{6\}$$
$$dom(z) = \{13\}$$

The domains are now bound consistent since $1 + 6 + 13 = 20$. In fact, they are also arc consistent in this case.

Range Consistency

The definition of the range consistency is similar to the bound($\mathcal{Z}$) consistency. Both look at the intervals instead of the domains. The difference here is that the bound consistency only looks at the minimum and maximum values of a domain unlike the range consistency which looks at all the values of the domain. Hence, this is a definition for the range consistency :

Definition 11 *The domains of the variables are said to be range consistent according to a constraint $\mathcal{C}$ if and only if for every variable $\mathcal{X}_i$, assigning it to one of its possible values makes the constraint satisfiable while considering other variables' domains as intervals.*

$$\forall i \ \forall x \in dom(\mathcal{X}_i) : \mathcal{X}_i = x \ \& \ (\forall j \neq i : \ \mathcal{X}_j \in \mathcal{I}(dom(\mathcal{X}_j)))$$
$$\Rightarrow \ \mathcal{C} \ is \ satisfiable$$

2.1.2 Search

One of the most important aspects of a solver is the search it performs to find the solutions of a problem. Most solvers use a search tree. At each node they reach what is called the fix-point:

Definition 12 *A problem is at fix-point if there is no more possible reduction of the domains of the variables by propagating a constraint.*

A search begins at the root of the tree. To go from a parent node to one of its children, a new constraint is added to the current problem. This new constraint will permit to once again reduce the domains of the variables until a new fix-point is reached. Those added constraints are most of the time unary constraints (constraints over a single variable). Therefore, only one variable's domain is reduced by the added constraint. Here is an example of a parent node and its children.

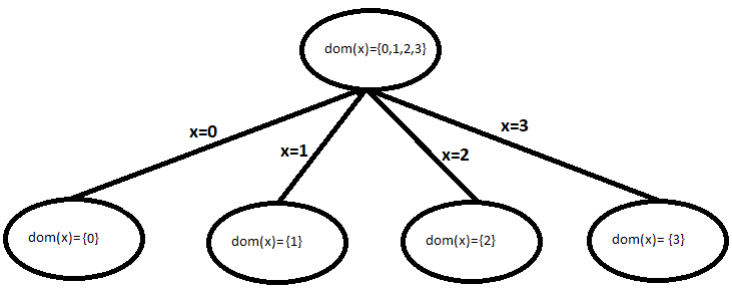


Figure 2.1: Example of a node and its children in a search tree in Constraint Programming.

Here, the variable x is the one that is reduced when going from the parent node to one of its children. The constraint used is $x = v$ when v is a value of its domain. Since there exists a child for each of the values of the domain of x, we know that by going through the children, we will not miss a solution of the problem described by the parent node.
In a search tree, a node is a leaf if it is a solution or if the problem has been detected as unsatisfiable.
A complete search tree should have all the solutions as leaves. Let's go over an example of a complete search tree. This search tree is about a problem composed of two variables x and y which both have the domain {1,2,3,4} and the constraint $x + y < 5$. The filtering algorithm used to propagate the constraint is arc consistent. Here is a possible search tree for this problem (Fig. 2.2).

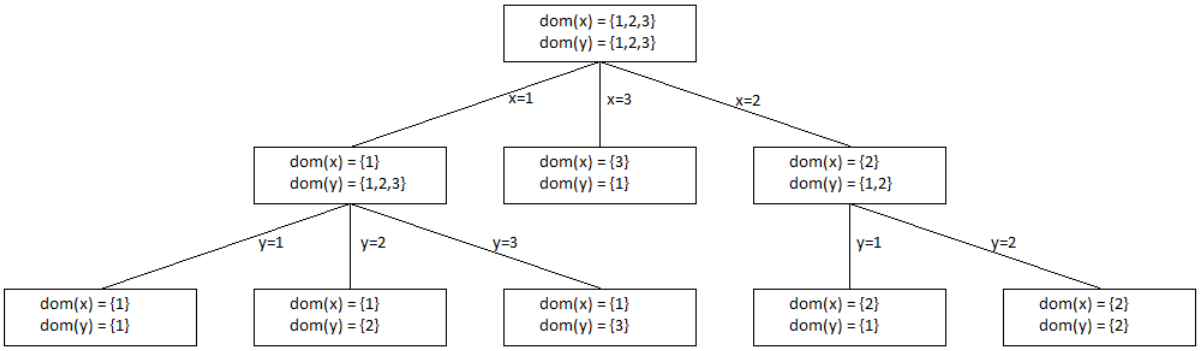


Figure 2.2: Example of a complete search tree

At the root of the tree, the domains are not the initial domains. At first the domains for x and y were both $\{1, 2, 3, 4\}$ but on the figure 2.2, they are both $\{1, 2, 3\}$. It is because as said before, a fix-point is reached at each node of the search tree. In fact, to be arc consistent, the domains of x and y both had to take off the value 4. At this point, propagating the constraint again would not change the result anymore. The fix-point is indeed reached.

To create the children, we arbitrarily choose to add constraints over x. Therefore, in each child node, a new fix-point must be reached considering this additional constraint too. In those children, the domain of x is always reduced to a single value because of the propagation of the new constraint. But only propagating this constraint is not always enough to reach the fix-point and the initial constraint can still reduce the domain of y if it is propagated. At this level, one of the children is a solution ($x = 3$ and $y = 1$) and therefore is a leaf. The other two can still have children and since y is the only not-fixed variable, the added constraint is over y. With the propagation of these new constraints, we obtain new solutions/leaves. There, all the six solutions of the problem are in the tree.

As seen in the example, the construction of a search tree needs choices such as which constraints are used to create the children of a node and over which variables. There already exists many methods to make the search tree smaller and the search faster. There are heuristics, local search, trailing, etc. This thesis will not go through all these techniques because it focuses on the constraints more than the search. Nevertheless, one of them is useful for this thesis and needs to be explained to better understand what we did : the trailing.

2.1.3 Trailing

One of the problems when searching using a search tree, is to backtrack to the parent node once one of its children has been visited. The easy way to do this would be to copy the parent node for each of its children to not have to backtrack and simply use the copies. But this method is

memory consuming. That is why trailing has become a good strategy for backtracking. Some articles explain in more details the trailing mechanisms such as [8]. Nevertheless, this thesis keeps an external look on the trailing mechanism.

A trailing algorithm uses a trail. This trail is based on a stack. Therefore, a trail has two operations: *Push* and *Pop*. Instead of copying the parent node, the state of the parent node is pushed on the trail (i.e. stack). Once a child has been visited, the last state pushed (which is the parent node) can be popped out of the trail. When it happens, the current state is restored to the last state pushed.

Here is a simple example of using trailing (Fig. 2.3). First, the states that are pushed and popped on this trail are the variables 'x' and 'y'. Each time the states of 'x' and 'y' are pushed on the trail, their states are saved in the stack of the trail and each time there is a pop, their states are restored to the last pushed state. The example is accompanied by its memory state to have a better understanding of the trailing.

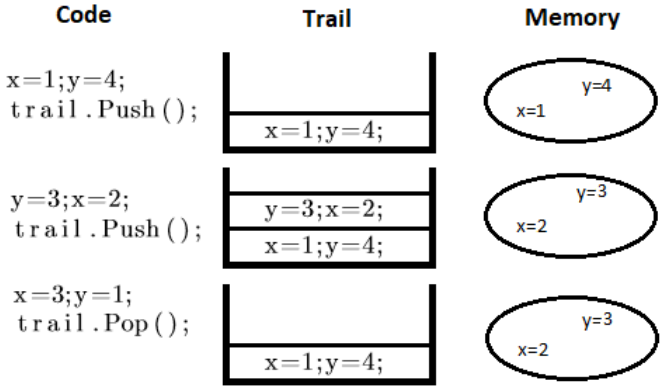


Figure 2.3: Example of a trailing with the code (on the left), the stack of the trail (in the center) and the memory (on the right).

In this example, you have first the variables x and y respectively set to 1 and 4. Then the state is pushed on the trail. This does not change the current state and in memory, $x = 1$ and $y = 4$ but the trail has now this state in its stack. After that, the current state is changed by modifying the values of x and y, then this new state is pushed on the trail. It does not erase the previous state pushed but simply goes on top of it. Now, the trail possesses two states in its stack. In the last lines, x and y are again set to a new value (3 for x and 1 for y). Nevertheless, this is not what is in memory. That is caused by the pop operation. It retrieved the last state pushed on its stack and restored the current state as this one. The last pushed state was when $x = 2$ and $y = 3$ therefore, those are the current values of x and y after the pop operation. Since that state has been popped out of the trail, it is also removed from its stack. This explains why the stack of the trail contains a single state in its stack instead of two.

In practice, some filtering algorithms evolve during a search. Therefore, to test and debug such filtering algorithms, it is important to test them while a search is performed. In this thesis, the principles of the trailing mechanisms are used to test those incremental filtering algorithms in the *Testing Incremental Filtering Algorithms* section.

2.2 Formalization of some Constraints of Constraint Programming

Constraint Programming is a very wide field, adapted to a big number of problems. In order to model all these problems, a lot of constraints exist. In this section, we will explain you some of the most important constraints occurring in the modelling of many problems. Most of these definitions have been inspired from [9].

2.2.1 AllDifferent Constraint

The *AllDifferent* constraint is a very used constraint for modelling simple but also more complex problems. This constraint is easy to understand. Defining the *AllDifferent* constraint over a set of variables means that all these variables must be assigned to different values. In other words, no two variables can have the same value. Here is a definition of the *AllDifferent* constraint in mathematical terms :

$$AllDifferent(\mathcal{X}_0, \mathcal{X}_1, ..., \mathcal{X}_k) = \{(x_0, x_1, ..., x_k) | \ \forall i \ \forall j \neq i \ x_i \in Dom(\mathcal{X}_i) \ , x_i \neq x_j\}$$

Considering for example three variables x, y and z such that $Dom(x) = \{1, 2\}, Dom(y) = \{1, 2\}$ and $Dom(z) = \{1, 2, 3\}$. Defining the constraint $AllDifferent(x, y, z)$ would impose that z could not be equal to 1 or 2. Indeed, since $Dom(x) = Dom(y) = \{1, 2\}$, setting z to 1 or 2 would lead to a dissatisfaction of the constraint, since then, there will remain only the same possible value for x and y, so they can't have different values. The possible instantiations of these variables satisfying the constraint are $x = 1, y = 2, z = 3$ or $x = 2, y = 1, z = 3$.

2.2.2 Sum Constraint

The *Sum* constraint is a basis of Constraint Programming. This constraint contains additional parameters that are the relation to be considered (lesser than, greater than, equal, ...) and the constant. Defining a *Sum* constraint over a set of variables simply imposes that the sum of the variables must respect the relation with respect to the constant. Here is a definition for the *Sum* constraint using mathematical terms :

$$Sum(\mathcal{X}_0, \mathcal{X}_1, ..., \mathcal{X}_k) \ R \ c =$$

$$\{(x_0, x_1, ..., x_k) | \ \forall i \ x_i \in Dom(\mathcal{X}_i), \sum_{i=0}^{k} x_i \ R \ c \text{ where } c \text{ is a constant and } R = \{<, >, =, \leq, \geq\}\}$$

Considering for example three variables x, y and z such that $Dom(x) = \{1, 2\}, Dom(y) = \{1, 2\}$ and $Dom(z) = \{1, 2, 3\}$. Defining the *Sum* constraint $sum(x, y, z) < 5$ would impose that z could not be equal to 3. Indeed, setting z to 3 would necessarily lead to a sum that is greater or equal to 5, since $x \geq 1$ and $y \geq 1$. A possible instantiation of the variables satisfying the constraint could be $x = 1$, $y = 2$, $z = 1$.

2.2.3 Element Constraint

The *Element* constraint has three different kinds of variables : an array $\mathcal{X}$, an index $\mathcal{I}$ and a value $\mathcal{V}$. Each item of the array is a variable in itself and therefore has a specific domain, idem for the index and value variables. When defining the *Element* constraint over these variables, it states that the ith variable of the array named x_i must be equal to the value v. Here is a definition of the *Element* constraint using a mathematical notation :

$$Element(\mathcal{X}, \mathcal{I}, \mathcal{V}) = Element([\mathcal{X}_0, \mathcal{X}_1, ..., \mathcal{X}_k], \mathcal{I}, \mathcal{V}) =$$

$$\{(x_0, x_1, ..., x_k, i, v) | \ \forall j \ x_j \in Dom(\mathcal{X}_j), i \in Dom(\mathcal{I}), v \in Dom(\mathcal{V}), x_i = v\}$$

Considering now an array a containing three variables having respectively domains {1,2} {4,5} {3,5}, a variable i having as domain {0,1,2} and a variable v in {1,3}. With all these domains, we can see that the variable i cannot be equal to 2 because the domain of $a_2 = \{4,5\}$ contains no possible value of v. A possible instantiation of the variables satisfying the constraint is $a_1 = 1$, $a_2 = 4$, $a_3 = 5$, $i = 0$ and $v = 1$.

2.2.4 Table Constraint

The *Table* constraint considers an array of variables and has a table as an additional parameter. The table contains possible instantiations for the variables in the array. The constraint simply states that the instantiation of the variables of the array must belong to the table. Here is a formulation of this *Table* constraint in mathematical terms :

$$Table([\mathcal{X}_0, \mathcal{X}_1, ..., \mathcal{X}_k], \mathcal{T}) = \{(x_0, x_1, ..., x_k) |\ \forall i\ x_i \in Dom(\mathcal{X}_i), (x_0, x_1, ..., x_k) \in \mathcal{T}\}$$

Consider as example an array containing two variables x and y with domains {1,2,3} and the table represented below :

x	y
1	1
1	2
2	3
2	1

Since x and y cannot take values that are not present in the table, we can immediately say that x cannot be equal to 3. This is the only value we could remove from the domains without removing a solution. The possible instantiations of the variables satisfying this *Table* constraint are all the instantiations of x and y that are present in the table ([1,1], [1,2], [2,3] and [2,1]).

2.2.5 Circuit Constraint

The *Circuit* constraint considers an array of variables whose domains are included in the indices of this array. An instantiation of the variables respecting the constraint can be seen as a circuit beginning at the index 0. Then the value of the variable at that position will give the next checkpoint (index) the circuit should go through. The variable at this checkpoint will again indicate where is the next checkpoint (index). To respect the *Circuit* constraint, the circuit described by the instantiation must pass by all the checkpoints and then return at its beginning to close the circuit.

To mathematically formulate this constraint, the notion of cyclic permutation must be defined. Let S be a permutation $(s_1, \ldots, s_N)$ of $(1, \ldots, N)$. The circuit C_S of a permutation S is defined as follow:

$$\begin{aligned} &1 \in C_S \\ &i \in C_S \Rightarrow s_i \in C_S \end{aligned}$$

S is a cyclic permutation if its circuit is of size N. This definition exactly represents the circuit presented before with its checkpoints.
With this definition of cyclic permutation, it is easy to give a mathematical expression for the *Circuit* constraint.

$$\begin{aligned} Circuit(\mathcal{X}_0, \ldots, \mathcal{X}_N) = \{(x_0, \ldots, x_N) | \forall i\ x_i \in dom(\mathcal{X}_i), \\ (x_0, \ldots, x_N)\ \textit{is a cyclic permutation of}\ (0, \ldots, N)\} \end{aligned} \tag{2.1}$$

The equation 2.1 comes from [9] with its formal definition of cyclic permutation.

Here is an example of a circuit problem. Let 3 variables x,y and z have the respective domains : $\{0, 2\}$, $\{0, 1, 2\}$ and $\{1, 2\}$. Here, we will try to find an instantiation from those domains satisfying the *Circuit* constraint. First, we look at x. Suppose the variables are in lexical order in an array. x is at the index 0. If $x = 0$, the circuit will be closed without passing by y or z. Therefore, x cannot be 0. Let's try to form a circuit with $x = 2$. Since $x = 2$, let's look at the variable at that index : z. If $z = 2$, the circuit will infinitely loop over the index of z. So, z cannot be 2. Then, let's try to form our circuit with $z = 1$. The next variable we look at is thus the one at index 1, which is y. There, all the variables have been visited by our current circuit. Thus, now, the circuit must be closed. y can be 0, 1 or 2. Those three values both permits to close the circuit but the definition of circuit says that we need a permutation. The value 2 and 1 have already been taken by x and z, so y can only be 0. In fact, for the *Circuit* constraint, the circuit must be closed by returning to its beginning. We could have directly said that y could only be 0 since it was the index where we began the circuit. Finally, the circuit $(2, 0, 1)$ is correct. In fact, it is the only circuit existing for this problem.

2.2.6 Global Cardinality Constraint

The global *Cardinality* constraint considers an array of variables and additional variables defining the number of times given values $\{v_0, v_1, ...v_j\}$ can occur in the array. So, it considers an array of variables $\mathcal{X}$ each having its own domain and additional variables $\mathcal{C}_{v_i}$. The constraint states that the value v_i must occur $\mathcal{C}_{v_i}$ times in $\mathcal{X}$. More formally, the constraint can be defined as :

$$Cardinality(\mathcal{X}, \mathcal{C}) = Cardinality([\mathcal{X}_0, \mathcal{X}_1, ..., \mathcal{X}_k], [\mathcal{C}_{v_0}, \mathcal{C}_{v_1}, ..., \mathcal{C}_{v_j}])$$
$$= \{(x_0, x_1, ..., x_k, c_0, c_1, ..., c_j) | \forall i \; x_i \in Dom(\mathcal{X}_i), \forall l \; c_l \in Dom(\mathcal{C}_{v_l}), occ(v_l, (x_0, x_1, ..., x_k)) = c_l\}$$

where $occ(v, \mathcal{X}) = c$ means that the value v occurs c times in the array $\mathcal{X}$.

Consider now an example of the global *Cardinality* constraint. Suppose we have an array with three variables x, y and z. Their domains are {1,2,3}. Suppose also we have the three count variables c_1 with domain {0,1}, c_2 with domain {1,2} and c_3 with domain {0,3}. Since we have three variables to assign to a value, assigning c_3 to 3 will force $c_1 = c_2 = 0$ which is impossible because of c_2. It forces $c_3 = 0$. From now, $c_1 + c_2$ must be equal to 3. So, $c_1 = 1$ and $c_2 = 2$. It results that, among x, y and z, two of them must be equal to 2 and one of them must have value 1. A possible solution is to set $x = 1$ and $y = z = 2$.

2.3 Concrete Examples of Constraint Programming Models

2.3.1 N-Queens Problem

The problem is the following : given a chessboard of size N*N, place N queens on it so that there can't be more than one queen in each column, line and diagonal. This problem can be modelled in the field of Constraint Programming as follows:

- a set of N variables $\mathcal{X}_i, 0 \leq i < N$ representing the column of the queen situated in row i. Each of these variables $\mathcal{X}_i$ has a domain $\mathcal{D}_i = \{0, 1, 2, ..., N - 1\}$
- a set of constraints :
  1. $allDifferent(\mathcal{X}_0, \mathcal{X}_1, ..., \mathcal{X}_{N-1})$
  2. $allDifferent(\mathcal{X}_0, \mathcal{X}_1 + 1, \mathcal{X}_2 + 2, ..., \mathcal{X}_{N-1} + N - 1)$

3. $allDifferent(\mathcal{X}_0, \mathcal{X}_1 - 1, \mathcal{X}_2 - 2, ..., \mathcal{X}_{N-1} - N + 1)$

Implicitly, from the variables' definition, the condition saying that there cannot be more than one queen in a row is respected, since $\mathcal{X}_i$ represents the position of the queen situated in row i. So, there is exactly one queen in each row.

The other conditions of the problem are formalized by the model's 3 constraints listed above. The constraint (1) simply formalizes the fact that there cannot be more than one queen in a column. The constraint (2) formalizes the fact that there cannot be more than one queen in the descending diagonals, and the constraint (3) does the same for the ascending diagonals. With all these constraints, the model is complete.

Here is an example for an 8-queens problem[1] :

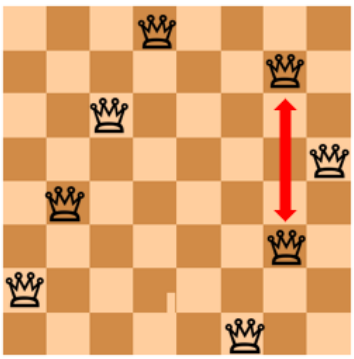


Figure 2.4: A rejected solution

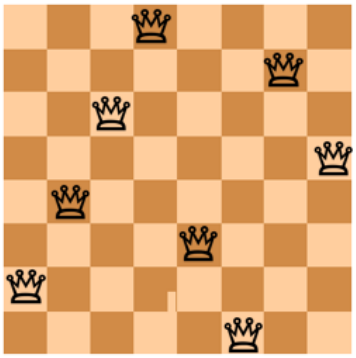


Figure 2.5: An accepted solution

The figure 2.4 is not a valid solution to the 8-queens problem because there are two queens in the seventh column. In terms of the model's variables, the variable $\mathcal{X}_1 = \mathcal{X}_5$, which is in contradiction with the constraint (1).

The figure 2.5 represents a valid solution : there are no column nor row nor diagonal with more than one queen in it and so all the defined constraints are respected.

2.3.2 Magic Square

The problem of the magic square [10] consists in placing the numbers from 1 to N^2 over a grid of size $N * N$, while respecting some constraints. The constraints state that each row, each column and the two main diagonals must have the same sum.

Let's formalize this problem using Constraint Programming terms :

- a set of N^2 variables $\mathcal{X}_{ij}, 0 \leq i, j < N$ representing the tiles of the grid. Each of these variables $\mathcal{X}_{ij}$ has a domain $\mathcal{D}_i = \{1, 2, 3, ..., N^2\}$. A sum variable S representing the sum of the rows, columns and diagonals.
- a set of constraints :
  1. $allDifferent(\mathcal{X})$

[1] These illustrations use the grid illustration of `https://upload.wikimedia.org/wikipedia/commons/thumb/d/d7/Chessboard480.svg/208px-Chessboard480.svg.png` and the queen illustration of `https://rosettacode.org/wiki/N-queens_problem`

2. $\forall i \ \sum_{j=0}^{N-1} \mathcal{X}_{ij} = S$
3. $\forall j \ \sum_{i=0}^{N-1} \mathcal{X}_{ij} = S$
4. $\sum_{i=0}^{N-1} \mathcal{X}_{ii} = S$
5. $\sum_{i=0}^{N-1} \mathcal{X}_{i(N-1-i)} = S$

The first constraint states that each number between 1 and N^2 appears exactly once. The second and third constraints ensure that the sum of the elements of each row/column equals to S. And the two last constraints state that the sum of the two main diagonals also equals to S. All these constraints are sufficient to model the magic square problem.

Here is an example considering a magic square for $N = 3$:

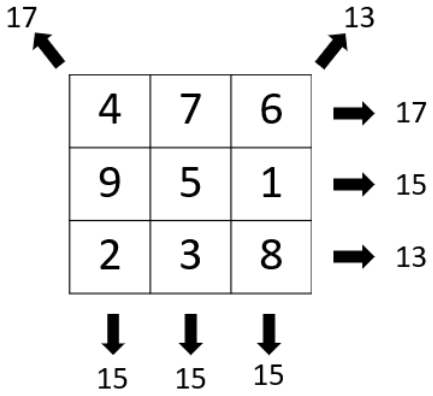


Figure 2.6: A rejected solution

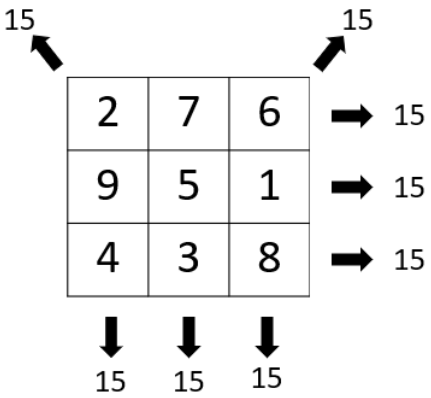


Figure 2.7: An accepted solution

On the figure 2.6, the sum of each column equals to 15 while the sum of the first and third rows is different from 15. Also, the sum of the elements on the diagonals is not the same either. So, this is not a valid solution.

Considering now the figure 2.7, the sum of each row, column or diagonal is equal to 15. This is a valid solution for the magic square problem with $N = 3$.

Chapter 3

CPChecker

This chapter will give an overview of the *CPChecker*'s features. This tool is able to test simple filtering algorithms reaching fix-point as well as more complex ones involving trailing mechanisms. All along the creation of *CPChecker*, we pay particularly attention to keep it as generic as possible. The major asset of the tool is that it is able to test the filtering implementation of any constraint acting over integer values. It has been written in *Scala* and has been made fully usable in *Java* too.

The first part of this chapter concerns the testing of algorithms reaching fix-point. The second part is about the testing of those algorithms during a search. After that, we will present *CPChecker*'s own assertion error implemented to ease the creation of unit tests.

3.1 Representation and Generation of the Variables

In order to test the filtering process of the variables, we generate many random domains. For this, we had to define conventions to represent variables in a way that is comprehensible by the user and easy to use. But one difficulty is that these domains have to be as diversified as they can, to cover many cases including more complicated ones for any constraint. This will be detailed in this section.

3.1.1 Variables Representation

The variables are represented by their domains. Indeed, the only elements identifying a variable are its domain and its identifier (unique name for each variable). So, we decided to represent the set of variables by an array of domains. Hence, the identifier of a variable is its index in the array.

In order to be generic and user-friendly, the domains are represented as sets of integers. Since there is no more than once the same value in a domain, sets are more pertinent than arrays or lists. Also, this representation does not require the creation of a specific class in *CPChecker*. This makes the implementation more bug-free. In addition, a user does not need to understand the implementation of that particular domain object that would have been created otherwise.

3.1.2 Domains Generation

In order to test the user's filtering implementation, random test domains are generated using the existing library *ScalaCheck*[11]. It is a library which is used to automate the testing of properties written in *Scala* or *Java*. In particular, we used *ScalaCheck* for the domains generation and the property testing.

Implementation principles

In terms of implementation, we generated random domains using the *ScalaCheck* functions `forAll` and `check`. When calling `check` on a property, *ScalaCheck* will generate random cases and will test this property over them, reporting any error found. When finding an error, *ScalaCheck* will try to reduce the test instance in order to create a more readable one on which it does not work. For reducing a test instance, *ScalaCheck* tries to remove some parts of the test instance and relaunches the property check over the reduced test instance. If it fails again, a test instance reduction has been found and *ScalaCheck* will try to reduce it even more. Otherwise, this part cannot be removed and it takes another part to be reduced.

In order to generate domains, we had to define random instances of domains, i.e arrays of sets of integers. For this, we chose to define ranges to which the variables domains belong. It was needed, because doing a pure random generation will not work. Consider for example the *AllDifferent* constraint. Taking for instance 10 variables of purely random domains will nearly always generate cases where no value can be filtered. So, there is no filtering and a function that simply returns the initial domains without doing any computation will do the trick and will be considered as a good filtering by our tool. This is something that should be absolutely avoided. So, we had to fix the range of the variables before calling the `check` function of *ScalaCheck*.

In order to fix the ranges, the *ScalaCheck* function `Gen.choose(min:Int, max:Int)` is used. This function simply returns a generator of random numbers between the `min` and `max` values included. With a bit of reasoning and implementation based on it, we created random domains instances.

From this, a problem remained. Since *CPChecker* is generic, it can be used to test any constraint. With this in mind, the same generator can't be used to test with relevance any constraint. Considering again the *AllDifferent* constraint, having domains with too many possible values will not be relevant because then, no value would be removed from the domains, as explained earlier. So, considering 10 variables varying in [-10,10] could be a good choice. However, considering the constraint saying that $\sum_{i=0}^{n} x_i > 1000$ with 10 variables between [-10,10] will always be unsatisfiable. So, a filtering function returning always empty domains should be considered as correct for this constraint. This is to be avoided.

To solve this, we decided to make the domains generator modifiable by the user. For this, we created a `TestArgs` class representing the test parameters that can be modified by the user. It contains all information about the test parameters with setters giving the possibility to the user to modify each of these pieces of information.

To make the test parameters modifiable by the user, we had to enter in the heart of *ScalaCheck* to understand how to change some parameters such as the number of tests created when calling *ScalaCheck*'s `check` function. There, we noticed that there is a `Test` module that is responsible for the execution of the tests in *ScalaCheck*.

This module has a trait named `Test.Parameters` that contains parameters such as the number of tests to execute. So, we created an inner class in the `TestArgs` class extending the trait `Test.Parameters`. In this inner class, we are allowed to change the number of tests that are made when applying the `check` function. Passing an instance of this class to the *ScalaCheck*'s `check` function generates the desired number of tests.

The `TestArgs` class is also responsible of other modifications that can be made to the generator, such as the change of the number of variables, their ranges, ... The next section will

explain in more details how the user can change these parameters.

User's perspective

By default, the number of generated test instances is set to 100 and each of these instances contains 5 variables with domains of 4 values. The domains contain by default random values between the range [-10,10]. As explained earlier, all these parameters can be adapted by the user in order to fit well with the constraint he wants to filter.

To change the parameters of the tests, the user has to create a new object of type `TestArgs` and simply call its functions. So, to change the number of generated test cases, the user can do the following :

```
val testArguments: TestArgs = new TestArgs()
testArguments.setNbTests(120)
```

These lines of code simply create new test arguments with all default parameters except the number of generated test cases that is set to 120.

Similarly, the user can adapt the number of variables, the sizes of the domains and their ranges. Note that for the sizes of the domains and their ranges, the user can either choose to change it for one variable or for all :

```
val testArguments: TestArgs = new TestArgs()
// set the number of variables to 10
testArguments.setNVar(10)
// set the domains of all variables within range [-5,5]
testArguments.setRangeForAll(-5, 5)
// set the domain of the first variable within range [-2,2]
testArguments.setRange(0,(-2,2))
```

In this little example, new parameters are created. Then, the number of variables is changed for considering 10 variables instead of 5 (default value). After that, the range of all variables is modified to make them vary between -5 and 5. Finally, the range of the first variable is changed to make it vary between -2 and 2.

Note that the order of these lines has its importance. Indeed, when changing the number of variables to a given number n, we will then generate n variables with the default range. This range is [-10,10] but if the user decides to set the range of all the variables to a new range, this default range is changed to the new range. Let's see an example of this :

```
val testArguments: TestArgs = new TestArgs()
// set the domains of all variables within range [-5,5]
testArguments.setRangeForAll(-5, 5)
// set the domain of the first variable within range [-2,2]
testArguments.setRange(0,(-2,2))
// set the number of variables to 10
testArguments.setNVar(10)
```

Now, all the 10 variables vary between [-5,5]. Indeed, first, all the variables have their range set to [-5,5]. The `setRangeForAll` also set the default range to [-5,5]. Then the variable at index 0 has his range set to [-2,2]. But then, the `setNVar` function will set the generator to 10 variables with the default range and density. Therefore, in the end, all the variables have a range of [-5,5].

In order to change the size of the domains, we use the notion of *density*. The *density* refers to the number of different values in the domains over the size of the range. For example, considering a variable x with its range of [0,9] and the domain $\{5, 7, 9\}$, the density is : $density(x) = \frac{|\{5,7,9\}|}{|[0,9]|} = \frac{3}{10} = 0.3$. This *density* can be changed by the user in the same way that for the range :

```
val testArguments: TestArgs = new TestArgs()
// set the densities of all variables to 0.8
testArguments.setDensityForAll(0.8)
// set the density of the first variable to 0.5
testArguments.setDensity(0,0.5)
```

The user also has the opportunity to set a seed to the generator in order to generate the same sequence of domains. It is particularly useful for him during the debugging phase to have the ability to redo exactly the same tests. Here is the line of code to set a seed :

```
val testArguments: TestArgs = new TestArgs()
testArguments.setSeed(100)
```

This concludes the part explaining the generation of the test instances on which the user's implementation will be tested.

3.2 Testing Static Filtering Algorithms

The idea of this part is to provide to the user a way to test the filtering of the domains of any constraint while reaching a fix-point. In other words, providing a way to test that a constraint reduces adequately the domains during the propagation towards the fix-point.

We will neither test the trailing mechanisms, nor the evolution of the internal variables. This will be tested in the *Testing Incremental Filtering Algorithms* part. Here, we will only check that given random input domains, the output domains are correctly reduced by the filtering algorithm according to some specifications. So, any user willing to test and debug his filtering algorithm can use *CPChecker*.

To test a filtering algorithm, two filtering algorithms must be considered: the tested filtering and the trusted filtering serving as reference for the testing. The trusted filtering should be considered as correct (bug-free) by the user. Each of these filtering algorithms takes domains as argument and returns these domains but filtered.

The process of testing a filtering algorithm can then be divided into those few steps :

1. Domains generation : some random domains are generated. They will be used to test the user's implementation. Note that this generation is configurable (a seed can be set, the number of variables can be changed, etc). For more information about the configuration, see section 3.1.2.
2. Application of the tested filtering algorithm with these random domains.
3. Application of the trusted filtering algorithm with the same inputs as the tested filtering algorithm.
4. Comparison of the two filtering algorithms. This means making the comparison of the filtered domains generated at step 2 and the ones generated at step 3.

There is below an illustration of the whole testing process (Fig. 3.1) for one random generation of domains. This process will be repeated by default 100 times.

As explained earlier, a *filtering algorithm* refers here to the function that given input domains, returns the filtered domains by the filtering procedure defined by the algorithm.

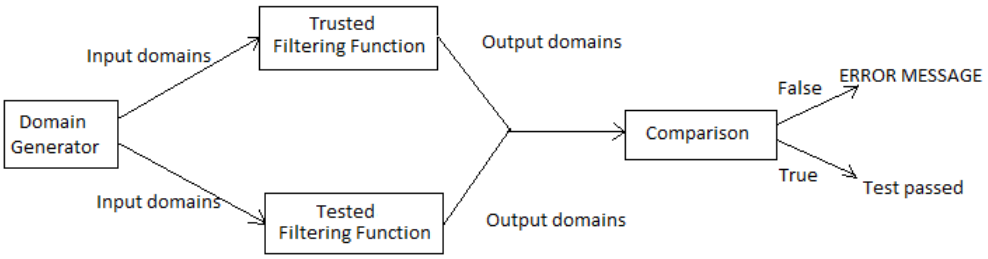


Figure 3.1: Schematic representation of the testing of one generation of domains.

3.2.1 Equality of Filtering Algorithms

CPChecker allows the user to check that two filtering algorithms reduce domains in the same way over the random test instances. So, it will generate random test instances as explained in the previous section, and after the application of the two algorithms, it will check that the filtered domains are the same.

In terms of implementation, the function that is responsible of the whole testing procedure is the `check` function of the `CPChecker` object. This `check` function will test the equality of the results of both algorithms over the test instances and will generate output documents with the results of the tests.

Here is the signature of this `check` function :

```
def check(bugFreeFiltering: Filter, testedFiltering: Filter)
        (implicit testArguments: TestArgs, stats: Statistics)
        : Boolean
```

The first argument is the `bugFreeFiltering` which is the filtering the user assumes as correct. It can either be *CPChecker*'s filtering implementation or another filtering implementation trusted by the user. This `bugFreeFiltering` will be assumed without any bug and will serve as reference during the check process. The second argument `testedFiltering` is the filtering that will be compared with the `bugFreeFiltering`. Here, we will verify that the filtered domains of both algorithms are identical. The `implicit testArguments` simply refers to the test arguments that the user can modify if he wants to. The `implicit stats` argument refers to the statistics which is the object responsible of the output. This output will be detailed later in this thesis.

This function returns true if all tests are passed successfully, i.e if over all the generated domains, both filtering algorithms returns the same reduced domains. Otherwise, it return false.

Note that the two first arguments are of type `Filter`. The `Filter` class is an abstract class. To recall, an abstract class is a class that contains abstract methods, which means methods defined by their signature but that have no body. To instantiate an abstract class, these abstract methods must be implemented. Here, it contains the abstract `filter` method. This `filter` method takes in argument the variables' domains and returns the filtered domains. Here is the signature of such a function :

```
def filter(variables: Array[Set[Int]]): Array[Set[Int]]
```

Let's explain this function in more details by giving formal specifications :

- Preconditions : the input domains are represented by the type `Array[Set[Int]]`. Each entry of the array corresponds to the domain of a different variable. The number of considered variables is then defined by the length of the array. This array is not null and does not contain any empty domain.
- Postconditions : returns the filtered domains as an `Array[Set[Int]]`. If the filtering reveals no solution could be found given these domains, returns an array of empty sets or throws a `NoSolutionException`.

In the case where the filtering algorithm finds there is no possible solution given the domains, this `filter` function allows two behaviors. Either returning an array of empty domains or throwing a `NoSolutionException`. The `NoSolutionException` is a class created to deal with such situations. Note that in the case where this `filter` function returns an exception that is not a `NoSolutionException`, the exception will be caught and considered as no solution found, but it will be printed in the terminal in order to let the user know he throws a non expected exception.

Now let's see how this `check` function is implemented :

```
def check(bugFreeFiltering: Filter, testedFiltering: Filter)
         (implicit testArguments: TestArgs, stats: Statistics): Boolean = {
  var result: Boolean = true
  forAll(testArguments.gen) { x =>
    if ((x.length < testArguments.getNbVars) || checkEmpty(x)) true
    else if (!checkConstraint(x.toArray, bugFreeFiltering, testedFiltering,
                              comparisonCheck(_, null, stats))) {
      result = false
      false
    }
    else true
  }.check(testArguments.getTestParameters)
  stats.setGenerator(testArguments)
  stats.print
  result
}
```

This function takes two implicit parameters as arguments : `testArguments` and `stats`. It means that the user can call the `check` function without specifying them. When doing this, the parameters that will be used are the default ones.

We use the `forAll{...}.check` function of *ScalaCheck* in order to test the property within the `forAll`. Note that the `forAll` concerns the `testArguments.gen` that is the variable generator of the test arguments passed as implicit parameters. Into the `forAll`, three checks are made. The first condition is necessary because, as mentioned before, when finding a failing test, *ScalaCheck* will reduce the test case until finding a more readable test case for the user. But, for some constraints, we must be careful with this. Indeed, some variables can have particularities and are absolutely necessary for the constraint. It is the reason why we can't simply consider that all variables have the same importance and we can't remove whatever variable while reducing. To solve this, we will neglect *ScalaCheck* reductions removing variables. So, when the size of the `x` argument is lesser than the required number of variables imposed by the test parameters, we simply return true avoiding evaluating this non interesting test. The same applies in the case where the generated domains contain an empty domain. It implies directly there is no solution and can cause error if the user doesn't take that into account. This is why the `checkEmpty` function has been created. It returns true if one of the variables' domain is empty and false otherwise.

One can think that, by adding this first condition, we do not consider any *ScalaCheck* reduction anymore. But this is not the case. *ScalaCheck* makes reductions by removing a whole variable domain, but it also makes reductions of some values of a given domain. These reductions are still valid and will be considered by the `check` function. So, we still make test cases reductions allowing the user to have more readable examples, but we do not consider reductions involving the removing of a whole variable domain.

On the remaining relevant tests, the `checkConstraint` function is called. It will apply the two algorithms and will check they return the same filtered domains. Notice that the last argument of the `checkConstraint` function is the function which compares the filtered domains. Finally, the statistics' generator is set for the output and the `print` method is called to display it.

3.2.2 Comparing Strength of Filtering Algorithms

The user can determine if one filtering algorithm filters more than another thanks to the *CPChecker*'s `stronger` function. This can be useful to compare filtering algorithms.

Here is the code of the `stronger` function :

```
def stronger(strongerFiltering: Filter, filtering: Filter)
      (implicit testArguments: TestArgs, stats: Statistics): Boolean = {
  var result: Boolean = true
  forAll(testArguments.gen) { x =>
    if ((x.length < testArguments.getNbVars) || checkEmpty(x)) true
    else if (!checkConstraint(x.toArray, strongerFiltering, filtering,
                              comparisonStronger(_, null, stats))) {
      result = false
      false
    }
    else true
  }.check(testArguments.getTestParameters)
  stats.setGenerator(testArguments)
  stats.print
  result
}
```

This function is very similar to the `check` function. The only difference lies in the domains comparison. Indeed, in the `check` function, the comparison requires the equality of domains while the `stronger` function requires that the filtered domains of the trusted filtering algorithm are included into the filtered domains of the tested filtering algorithm. This is reflected by the fact that the last argument of the `checkConstraint` function is a different function of comparison than for the `check` function.

Note that a particular way of using the `stronger` function could be to check that the tested filtering algorithm does not remove any solution. It is due to the fact that, calling `stronger` to compare the tested filtering with a bug-free filtering algorithm performing arc consistency, will pass the tests only if it does not remove any solution over all the test instances. Indeed, since the arc consistency reduces domains until there only remains solution values, removing more values than in the arc consistent filtering consists in removing solutions. So, to test that a filtering algorithm does not remove any solution, the user can compare it with a bug-free arc consistent filtering algorithm.

3.2.3 Testing Different Propagation Types

Sometimes, instead of comparing two algorithms, it is more interesting to look at the propagation property of a single filtering algorithm. Does it reach arc consistency? or bound consistency? or even range consistency? *CPChecker* provides an easy way to do this.

As seen before, two filtering algorithms can be compared as `Filter` objects with the `check` function. *CPChecker* possesses classes which extend the `Filter` class. Each of them defines a given consistency (arc consistency, bound consistency or range consistency) for any constraint.

Nevertheless, those classes are not all magical. It is impossible to filter correctly every constraint without even knowing which constraint is propagated. Therefore, those classes take as argument a checker function in their constructor. This checker function takes as argument an instantiation of variables (an array of integers) and must tell if it respects the constraint this function checks. For example, for the constraint $\sum x = 20$, the checker could be like this:

```
1 def sumChecker(x: Array[Int]): Boolean = x.sum == 20
```

To ease the usage of those classes, for some constraints, a checker function has been implemented by *CPChecker*. These functions are located in the `Checkers` object. But the number of existing constraints is far from being able to be counted on one's fingers. Hence, for most constraints, it must be implemented by the user.

Now, let's see *CPChecker*'s `Filter`'s one by one.

Arc Consistency

The definition of arc consistency implies reducing all domains to values belonging to solutions. There exists many elaborated algorithms performing that. But since this arc consistency implementation will be used as a correct reference to be compared with the other filtering implementations, it has to be bug-free. Indeed, we should never say to the user that his filtering does not reach arc consistency if it does.

One way to be bug-free for arc consistency is to create all solutions given the constraint. It means that we will create all variables' instantiations by doing a cartesian product. Then the ones which do not satisfy the checker function will be removed. The surviving instantiations are solutions since they respect the constraint.

Every element of an arc consistent domain is part of a solution. The reverse is also true: all the values of a solution are part of the arc consistent domains. Therefore, a variable's arc consistent domain is composed of all its values belonging to the solutions. Unfortunately, it will impose a restriction on the domains sizes that can be tested since taking too big domains will take a lot of time to execute.

The class that implements the `filter` method filtering domains to make them arc consistent has been called `ArcFiltering`. In this class, the implementation of the `filter` function looks like this :

```
override def filter(variables : Array[Set[Int]]) : Array[Set[Int]] = {
  if(variables.isEmpty) throw new NoSolutionException
  val solutions: Set[Array[Int]] =
    variables.foldLeft(Set[Array[Int]](Array()))((acc, x) => {
      var set: Set[Array[Int]] = Set[Array[Int]]()
      x.foreach(elem => acc.foreach(y => set += y :+ elem))
      set
    }).filter(x => checker(x))
  if (solutions.isEmpty) throw new NoSolutionException
  toDomains(solutions)
}
```

This algorithm is quite straightforward. First, we create the solutions by using the *Scala* function `foldLeft`[1]. This function takes in argument the initial value of the accumulator which is here a set of empty arrays. Then, we create the solutions by constructing them element by element. The solutions are built from the domains like this :

- Initial domains : $x_0 = \{1, 2, 3\}, x_1 = \{4, 5\}$, Initial `acc` $= \{[]\}$
- After step 1 : `acc` $= \{[1], [2], [3]\}$
- After step 2 : `acc` $= \{[1, 4], [2, 4], [3, 4], [1, 5], [2, 5], [3, 5]\}$

This illustrates how all the solutions are built. When all the solutions have been formed, we call the *Scala*'s `filter` function with the `checker` function provided by the user. This will remove the instantiations which are not solutions according to the `checker` function.

After that, if no solution remains, an array of empty sets is returned. Otherwise, the arc consistent domains are obtained from these solutions. This is done by calling the `toDomains` function :

```
def toDomains(solutions: Set[Array[Int]]): Array[Set[Int]] = {
  val variables: Array[Set[Int]] = Array.fill(solutions.head.length)(Set.
   empty)
  solutions.foreach { sol =>
    for (i <- variables.indices) {
      variables(i) += sol(i)
    }
  }
  variables
}
```

This function first creates `variables` as an array of empty sets. Then, for each solution, it decomposes it by putting each element of the solution into the corresponding variable's domain. And then, it returns the `variables`.

Bound($\mathcal{Z}$) Consistency

The `BoundZFiltering` class represents the filtering which reaches bound($\mathcal{Z}$) consistency given its checker function. Contrarily to the arc consistency, simply obtaining all the solutions will not allow to find the bound consistent domains. In fact, some values of a bound consistent domain may not be part of a solution. Even while computing all the solutions obtained from the intervals instead of the domains, retrieving the bound($\mathcal{Z}$) consistent domains from them would not be a simple task. The objective of the `BoundZFiltering` class is to propose a trustable bound($\mathcal{Z}$) consistent filtering algorithm. Hence, this method of generating all the solutions does not fit the

[1]This function will step by step build the solutions based on an accumulator. For more information, see foldleft in the Scala API

needs since it would require a complex implementation.

The filtering algorithm of `BoundZFiltering` applies the definition of bound($\mathcal{Z}$) consistency : for every variable, the minimum and maximum values of its domain are included in a solution if we consider the other domains as intervals. Basically, the algorithm will look at each minimum and maximum. Then for each of them, it will generate all the instantiations until one is a solution according to the checker function. If no solution is found, then the definition of bound($\mathcal{Z}$) consistency is not respected and this minimum or maximum is not a part of the bound($\mathcal{Z}$) consistent domains. Like that, the domains will be filtered little by little until the definition of bound($\mathcal{Z}$) consistency holds.

Here is the implementation of the `filter` function for the bound($\mathcal{Z}$) consistency.

```
override def filter(variables: Array[Set[Int]]): Array[Set[Int]] = {
  val intervals = variables.map(x => if (x.nonEmpty) new Interval(x)
                                    else throw new NoSolutionException)
  var changed: Boolean = true
  while (changed) {
    changed = false
    for (i <- intervals.indices) {
      if (changeBounds(i, intervals)) changed = true
    }
  }
  intervals.map(x => x.dom)
}

def changeBounds(i: Int, intervals: Array[Interval]): Boolean = {
  Array(intervals(i).min, intervals(i).max).foldLeft(false) {
    (acc,value) =>
      if (!findASolution(intervals, i, value)) {
        intervals(i).remove(value)
        true
      }else acc
    }
}
```

Here, the variables are first transformed into intervals. the `Interval` class is simple and is only use to ease the reading of the code. Here is the class:

```
class Interval(var dom: Set[Int]) {

  def min: Int = dom.min

  def max: Int = dom.max

  def getRange: Range = min to max

  def remove(i: Int): Unit = {
    if (dom.size == 1) throw new NoSolutionException
    dom = dom - i
  }
}
```

Then the bounds of each interval are checked with the `changeBounds` function. This function applies the definition of bound($\mathcal{Z}$) consistency. It looks at the minimum and maximum of the given interval. If they are not part of a solution (while looking at the intervals), they are removed. Once all the bounds have been looked at, if a value has been removed, then the definition of the bound($\mathcal{Z}$) consistency was not respected. Therefore, the `filter` function calls once again the

`changeBounds` function for each interval.

When a minimum (resp. maximum) is removed, the new minimum (resp. maximum) is obtained thanks to the domains and not the intervals. For example, for the domain $\{1, 5, 6\}$, its interval is $[1, 6]$. If the minimum is removed, the interval becomes $[5, 6]$. Therefore, the new minimum is 5 and not 2.

In the `changeBounds` function, to find if a minimum or maximum is part of a solution, the `findASolution` function is called. Let's see its implementation.

```
def findASolution(intervals: Array[Interval], index: Int, value: Int):
    Boolean = {
  val currentSol: Array[Int] = Array.fill(intervals.length)(0)
  //set the fixed value
  currentSol(index) = value

  def setIthVariable(currentIndex: Int): Boolean = {
    //skip the fixed value
    if (currentIndex == index) return setIthVariable(currentIndex + 1)
    //if the solution is complete, we check it with the checker
    if (currentIndex == intervals.length) return checker(currentSol)
    for (i <- intervals(currentIndex).getRange) {
      currentSol(currentIndex) = i
      if (setIthVariable(currentIndex + 1))
        return true
    }
    false
  }

  setIthVariable(0)
}
```

This function returns true if it finds a solution given a variable with a fixed value (here, it will be one of the bounds of its domain). It takes the intervals and two integers as arguments. Those two integers represent the value the solution it tries to find must contain. For that, its position in the solution and, of course, its value are passed in arguments. Hence, this function starts by creating the array representing the currently observed solution and sets the fixed value inside of it.

The current solution is then recursively created from its first value to its last with the `SetIthVariable` internal function. Then when the current solution is complete, it is checked with the checker function to verify it is truly a solution. Of course, when creating the current solution, the index of the fixed value is skipped to avoid rewriting it.

With that, the `findASolution` function has an easy to understand implementation and allows the implementation of the `filter` function of the `BoundZFiltering` class to have a generic and trustable bound($\mathcal{Z}$) consistent filtering algorithm.

Bound($\mathcal{D}$) Consistency

The `BoundDFiltering` class represents the filtering algorithm reaching the bound($\mathcal{D}$) consistency. Its implementation is similar to the bound($\mathcal{Z}$) consistency. The only difference is that domains are observed instead of intervals. Here is its implementation:

```
override def filter(variables: Array[Set[Int]]): Array[Set[Int]] = {
  var changed: Boolean = true
  while (changed) {
    changed = false
```

```
    for (i <- variables.indices) {
      if (changeBounds(i, variables)) changed = true
    }
  }
  variables
}

def changeBounds(i: Int, variables: Array[Set[Int]]): Boolean = {
  Array(variables(i).min, variables(i).max).foldLeft(false) {
    (acc, value) =>
      if (!findASolution(variables, i, value)) {
        if (variables(i).size == 1) throw NoSolutionException()
        variables(i) -= value
        true
      } else acc
    }
}
```

The names of the functions stay the same (`changeBounds` and `findASolution`) but the difference is that they take variables as arguments instead of intervals. The function `findASolution` has only one line which changes compared to the one for the bound($\mathcal{Z}$) consistency. It is in the `setIthVariable` internal function which instead of looking at all the elements of an interval looks at all the elements of a domain (set of integers).

```
def findASolution(variables: Array[Set[Int]], index: Int, value: Int):
    Boolean = {
  val currentSol: Array[Int] = Array.fill(variables.length)(0)
  currentSol(index) = value

  def setIthVariable(currentIndex: Int): Boolean = {
    if (currentIndex == index) return setIthVariable(currentIndex + 1)
    if (currentIndex == variables.length) return checker(currentSol)
    for (i <- variables(currentIndex)) {
      currentSol(currentIndex) = i
      if (setIthVariable(currentIndex + 1))
        return true
    }
    false
  }

  setIthVariable(0)
}
```

Range Consistency

The range consistency is also similar to the bound($\mathcal{Z}$) consistency. The only difference is that instead of looking at the minimum and maximum values of a domain, it looks at all the values of the domain. Therefore, the filtering algorithm used for the range consistency works like the bound($\mathcal{Z}$) consistent one. This filtering algorithm is implemented by the `filter` function of the `RangeFiltering` class.

To implement this filtering algorithm, the variables domains should be transformed into intervals. This part is the same as in the bound($\mathcal{Z}$) consistency. Then instead of a `changeBounds` function, it is the `filterInterval` function which is used. It is similar but instead of looking at the minimum and maximum, all the values of the domain are observed. Here is the code for the `filter` function of the `RangeFiltering` class:

```
override def filter(variables: Array[Set[Int]]): Array[Set[Int]] = {
  val intervals: Array[Interval] = variables.map(x => if (x.nonEmpty)
    new Interval(x) else throw new NoSolutionException)
  filterIntervals(intervals)
  intervals.map(x => x.dom)
}

def filterIntervals(intervals: Array[Interval]): Unit = {
  if (intervals.indices.foldLeft(false) { (acc, x) =>
    if (filterInterval(x, intervals)) true else acc })
    filterIntervals(intervals)
}

def filterInterval(index: Int, intervals: Array[Interval]):
Boolean = {
  val domain: Set[Int] = intervals(index).dom
  intervals(index).dom = domain.filter(findASolution(index, intervals, _))
  if (intervals(index).dom.isEmpty) throw NoSolutionException()
  domain.size != intervals(index).dom.size
 }

```

As for the bound($\mathcal{Z}$) consistency, all the intervals are filtered again and again until no intervals can be filtered anymore.

To know if a value of the domain being observed must be removed, the `findASolution` function of the bound($\mathcal{Z}$) consistency part has been used. In fact, that function does not care if the value is a minimum, a maximum or a value in-between. Hence, with this function and the *Scala*'s `filter` function, all the values that are not part of a solution are removed from the domain.

3.2.4 Another Way of Filtering

All these filtering methods are interesting but they only filter an instantiation violating the constraint when this instantiation is completely built. As a consequence, it necessarily computes all instantiations, even if a lot of them do not satisfy the constraint modelled by the checker.

Some checkers have a particular property : they are anti-monotonic functions. According to [12], an anti-monotonic boolean function can be defined like this :

Definition 13 *A boolean function f is anti-monotonic iff*

$$\forall Y \subseteq X, f(X) \Rightarrow f(Y)$$

From this, we can define the following property over an anti-monotonic boolean function :

Definition 14 *If a boolean function f is anti-monotonic, then*

$$\forall Y \subseteq X, \neg f(Y) \Rightarrow \neg f(X)$$

An example of such an anti-monotonic function can be a checker representing the *AllDifferent* constraint. It returns true if all the values passed as arguments are different and false otherwise. Considering any instantiation of n variables violating the *AllDifferent* constraint, nobody can find an extension of it that does not violate the constraint (if two values of the instantiation are the same, they will stay the same in the extension). For example, any extension of the instantiation [2,2] will always violate the *AllDifferent* constraint. But other checkers such as the one for the *Sum* constraint do not necessarily possess this property. Indeed, considering for

example $\sum x = 5$, the partial instantiation [0,1] does not satisfy the constraint, but the extended instantiation [0,1,4] satisfies it.

We noted that for anti-monotonic checkers, we can remove partial instantiations violating the checker during their construction. For this, we created new classes for each type of consistencies we are testing : `ArcPruning`, `BoundZPruning`, `BoundDPruning` and `RangePruning`. We called them `Pruning` because some part of the search space is pruned by stopping constructing complete instantiations when the partial ones already violate the checker.

ArcPruning

The `ArcPruning` class has a particular implementation performing the arc consistency while avoiding to create all solutions. In fact, this implementation is very similar to the one of the `ArcFiltering` class. The only difference lies in the fact that we make more calls to the `checker` functions for removing invalid instantiations during their construction. So, each time we add elements to the set of instantiations, we make a call to the `checker` function. Of course, the algorithm is still bug-free since it still constructs all the solutions but it filters the ones that do not respect the constraint earlier. But the user has to pay attention that this kind of filtering can only be used if the checker function is anti-monotonic. Here is the code of our `filter` function of the `ArcPruning` class :

```
override def filter(variables: Array[Set[Int]]): Array[Set[Int]] = {
  if(variables.isEmpty) throw new NoSolutionException
  val solutions: Set[Array[Int]] =
    variables.foldLeft(Set[Array[Int]](Array()))((acc, x) => {
      var sol: Set[Array[Int]] = Set[Array[Int]]()
      x.foreach(elem => acc.foreach(y => sol += y :+ elem))
      sol.filter(x => checker(x))
    })
  if (solutions.isEmpty) throw new NoSolutionException
  toDomains(solutions)
}
```

Here, this function will not be explained in details because it is similar to the function of the `ArcFiltering` from the section 3.2.3. The only difference is that the `sol.filter(...)` has been moved into the `foldLeft statement` in order to already filter partial instantiations.

BoundZPruning, BoundDPruning and RangePruning

The `BoundZPruning` and `RangePruning` classes contain their own implementation of the `filter` method that is very similar to their *Filtering* counterparts but that filters partial instantiations. In terms of implementation, the only difference is located in the `setIthVariable` internal function of the `findASolution` function :

```
def setIthVariable(currentIndex: Int): Boolean = {
  if (currentIndex == index) return setIthVariable(currentIndex + 1)
  if (currentIndex == intervals.length) return checker(currentSol)
  for (i <- intervals(currentIndex).getRange) {
    currentSol(currentIndex) = i
    if(checker(currentSol.take(currentIndex+1)))
      if(setIthVariable(currentIndex + 1))
        return true
  }
  false
}
```

The difference is that we added an additional `if` condition in the `for` loop in order to stop building the instantiation if the `checker` already finds that the partial instantiation violates the constraint (does not respect the `checker`).

For `BoundDPruning`, the change is the same but in its own implementation of the `findASolution`.

3.2.5 Combining Consistencies

A lot of constraints have variables representing different things. For example, the *Element* constraint has a variable representing an index, one representing a value and all the others representing elements of an array. Those differences between the variables have lead efficient filtering algorithms to filter those variables differently. Therefore, a filtering algorithm could, for example, filter some domains until they are arc consistent while others are range consistent. A filtering algorithm could reach different consistencies for different variables. This is why the `HybridFiltering` class has been created.

The `HybridFiltering` class takes in its constructor a checker function and an array of integers. The checker function is the same as for the arc, bound($\mathcal{Z}$), bound($\mathcal{D}$) and range filterings. The array of integers has each of its elements representing the filtering of the variables. The first element of the array tells which consistency should be reached for the first variable, the second tells for the second variable, etc. A 1 represents an arc consistent filtering, 2 a bound($\mathcal{Z}$) one, 3 a bound($\mathcal{D}$) one and 4 a range consistent one. All the other numbers told that no filters should be applied.

Let's see the implementation of this class.

```
1  class HybridFiltering(filterings: Array[Int], checker: Array[Int] =>
       Boolean) extends Filter {
2
3    val arc = new ArcFiltering(checker)
4    val boundZ = new BoundZFiltering(checker)
5    val boundD = new BoundDFiltering(checker)
6    val range = new RangeFiltering(checker)
7
8    override def filter(variables: Array[Set[Int]]): Array[Set[Int]] = {
9      val vars = variables
10     //The ac variables can directly be obtained thanks to the properties of
         arc consistency.
11     if (filterings.contains(1)) {
12       val acVars = arc.filter(variables)
13       for (i <- vars.indices) {
14         if (filterings(i) == 1) vars(i) = acVars(i)
15       }
16     }
17     // the BC and RC variable must recursively be reduced.
18     val intervals = vars.map(x => if (x.nonEmpty) new Interval(x)
19                                   else throw new NoSolutionException)
20     filterIntervals(vars, intervals)
21     intervals.map(x => x.dom)
22   }
23
24   def filterIntervals(vars: Array[Set[Int]], intervals: Array[Interval]):
       Unit = {
25     if (intervals.indices.foldLeft(false) { (acc, i) =>
26       if ((filterings(i) == 2 && boundZ.changeBounds(i, intervals)) ||
27         (filterings(i) == 3 && boundDChangeBounds(i, intervals)) ||
```

```
28          (filterings(i) == 4 && range.filterInterval(i, intervals)))
29          true
30        else acc
31      })
32        filterIntervals(vars, intervals)
33    }
```

This hybrid filtering uses the different filterings seen so far: `ArcFiltering`, `BoundZFiltering`, `BoundDFiltering` and `RangeFiltering`. Firstly, it applies arc consistency. Then it only keeps the filtered domains of the variables which should be arc consistent. This can be done since a domain is arc consistent if every value of the domain is part of a solution.

For the other consistencies, the variables are transformed into intervals then it filters once every interval considering the consistency it must reach. To filter them, the functions in the other filterings of *CPChecker* have been used. But for the bound($\mathcal{D}$) consistency, its `changeBounds` function takes variables instead of intervals as argument. Therefore, another function has been created to transform the intervals back into domains before calling the `changeBounds` function of the `BoundDFiltering`.

The `HybridFiltering` class, as the other filterings of *CPChecker*, possesses its pruning counterpart `HybridPruning`. the implementation of the `HybridPruning` class is the same as the `HybridFiltering` class. The only difference is that its global variables are instances of the `ArcPruning`, `BoundZPruning`, `BoundDPruning` and `RangePruning` classes instead of their filtering counterparts.

3.2.6 User Report

CPChecker has been created to help developers debugging the filtering algorithms. It generates random domains test cases and will check the user's filtering algorithm over these test cases. But since the tool is an help for debugging, when performing a set of tests, the user has to dispose of a way to see the kind of tests that have been realized and how the filtering algorithm faces these tests.

In order to create such a report, some statistics about the tests that have been realized are recorded during the check procedure. For this, the `Statistics` class was created to retrieve the useful information.

Calling one of the check functions (`check` or `stronger`) will automatically generate three statistics files. One of them, called *statistics.txt* contains all the general information about the proceeding of the tests. The two other files are named *passedTests.txt* and *failedTests.txt*. As their names suggest, the former contains the recording of all the tests that the tested filtering function passed with success, while the second contains the list of the tests that the tested filtering function did not perform correctly. It can be useful to record both the failed and the passed tests in order to allow the user to see what is the difference between these two kinds of tests and so, faster find his error. These three output files are located in a *out/statistics/* folder.

The *statistics.txt* file contains all general information over the executed tests. Here is an exhaustive list of the information it displays :

- The number of executed tests.
- The number of failed tests.
- The number of tests where the tested and trusted algorithms filter the same.

- The number of tests where the tested algorithm filters less than the trusted (trusted stronger than tested).
- The number of tests where the tested algorithm filters more than the trusted (tested stronger than trusted).
- The number of tests where the tested algorithm and the trusted filter differently (no stronger relation between the two).
- The number of tests where the trusted algorithm filters no value. This piece of information can be interesting because if the tests parameters are not well adapted to the constraint being tested, it is possible to have a lot of test cases where there is no value to filter. So, this is displayed to help the user judge of the pertinence of the generated test cases.
- The ratio of the number of tests where the tested and trusted algorithms filter until finding no solution over the number of tests where the trusted algorithm filters until finding no solution :

$$\frac{\#\ \textit{tests where tested and trusted algorithms filter domains until no solution}}{\#\ \textit{tests where trusted algorithm filters domains until no solution}}$$

  This ratio indicates to the user if the tested filtering detects there is no solution when it should.
- The ratio of the number of tests where the tested and trusted algorithms filter domains until obtaining a single solution over the number of tests where the trusted algorithm filters domain until obtaining a single solution :

$$\frac{\#\ \textit{tests where tested and trusted algorithms filter domains until a single solution}}{\#\ \textit{tests where trusted algorithm filters until a single solution}}$$

  This ratio gives an indication about the proportion of tests the tested filtering correctly filters until having a single solution.

There is also a display of the generator used. It allows the user to see the kind of domains on which these tests rely on. Here is an example of a *statistics.txt* file generated by comparing two filtering processes for the *Circuit* constraint (for *Circuit* constraint definition, see section 2.2.5). In these two algorithms, the trusted one should reach arc consistency and should be stronger than the tested one. This file has been created by calling the `check` function :

```
Here are the statistics of the executed tests :
-----------------------------------------

Number of executed tests : 87
Number of failed tests : 5

Tested and trusted algorithms filters the same : 82
Tested algorithm filters less than the trusted : 5
Tested algorithm filters more than the trusted (stronger) : 0
Trusted algorithm and tested one filters differently
(without one stronger than the other) : 0

Trusted algorithm filters no value : 0

Ratio of filtering until no solution found (tested algo/ trusted algo) : 41/41
Ratio of filtering until a single solution (tested algo/ trusted algo) : 28/30
```

```
Generator :
        Seed: 120
        variable        density         range
        0                       0.8             (0,4)
        1                       0.8             (0,4)
        2                       0.8             (0,4)
        3                       0.8             (0,4)
        4                       0.8             (0,4)
```

In this file, we can observe that 82/87 of the tests have been passed. We can also note that 82/87 of the test cases have been filtered in the same way by the tested and trusted algorithms. It seems coherent with the fact that, we require equality of the domains (by calling the `check` function), so we have the same number of passed tests than the number of tests where domains are equal. Also, in 5/87 of the test cases, the tested algorithm filters less than the trusted one. It corresponds to the 5 failing tests. This is expected since the tested algorithm does not filter until reaching arc consistency while the trusted algorithm does.

From the ratios, we can note that in 41 test cases, the trusted filtering algorithm finds there is no solution and in 30 of them, it filters until reducing the domains to a single solution. Over these 30 test cases, 28 have been filtered in the same way by the tested filtering algorithm.

Looking at the generator, it seems to be well chosen. The default generator has been slightly modified to be pertinent for the *Circuit* constraint. The range has been changed to be the variable indices, which must be the case for the *Circuit* constraint. Putting a bigger range would lead to more test cases where the trusted algorithm filters no solution, which is to be avoided. Also, the density has been increased in order to have all possible index values in each test case.

In the case where there are some errors, the user can look in the *failedTests.txt* file to understand the cause of the error. Here is an extract of this *failedTests.txt* file generated by the same call as for the *statistics.txt* file :

```
Initial domains   |Trusted filtering |Tested filtering
[3, 4]            |[4]               |[3, 4]
[0, 4]            |[0]               |[0, 4]
[1, 3]            |[3]               |[1, 3]
[1, 0]            |[1]               |[0, 1]
[2]               |[2]               |[2]

Initial domains   |Trusted filtering |Tested filtering
[0, 3, 4]         |[4]               |[3, 4]
[0, 2, 4, 3]      |[3, 0]            |[0, 4, 3]
[1, 3]            |[1, 3]            |[1, 3]
[1, 0]            |[0, 1]            |[0, 1]
[4, 2]            |[2]               |[2]
```

In this file, the `Initial domains` refer to the generated domains on which the filtering is tested. The `Trusted filtering` refers to the filtered domains by the trusted algorithm and the `Tested filtering` refers to the filtered domains by the tested algorithm. Each test case is separated by a blank line. Each line of a test case represents one variable domain. So,

in the first test case, there are initially 5 variables with the domains [3, 4], [0, 4], [1, 3], [1, 0] and [2].

The reason why these tests are recorded as failed tests is that the tested and trusted filtered domains are not the same for all variables. The tested filtering filters less than the trusted one. Note also that the first test is a reduction of the second one since when finding a failing test *ScalaCheck* tries to reduce it and we display first the most reduced tests. We decided to not only display the most reduced test because even though this is probably the most useful one, the other ones can contain additional information helping the user to find the source of the error.

Let's now look at the *passedTests.txt* file. Since here there are 82 tests successfully passed, this file is quite long. We will only display a small extract of it :

```
Initial domains   |Trusted filtering |Tested filtering
[0]               |[]                |[]
[0, 2, 4, 3]      |[]                |[]
[1, 3]            |[]                |[]
[1, 0]            |[]                |[]
[4, 2]            |[]                |[]

Initial domains   |Trusted filtering |Tested filtering
[4, 1, 3]         |[3, 4]            |[3, 4]
[0, 1, 2]         |[2, 0]            |[0, 2]
[3, 4, 2, 0]      |[4, 3]            |[3, 4]
[0, 2, 1]         |[1, 2, 0]         |[0, 1, 2]
[1, 2, 0, 4]      |[0, 2, 1]         |[0, 1, 2]
```

They are displayed in the same way that in the *failedTests.txt* file. Since the tests have been passed successfully, the `Tested filtering` will be exactly the same as the `Trusted filtering` every time.

But pay careful attention with this : when considering the `stronger` function instead of the `check` one, there may have tests that have been recorded as passed tests even though they do not have identical domains for both the `Trusted filtering` and the `Tested filtering`. Indeed, for the `stronger`, we do not require equality of both algorithms, but we require that the first one is stronger than the other one.

3.2.7 From Scala to Java

CPChecker has been written in *Scala*. So, what about other languages programmers willing to use the tool? Since *Scala* runs on the Java Virtual Machine (JVM)[2], the *CPChecker*'s bytecode can be understood from any other language built on top of the JVM.

But when calling a *Scala* function taking in arguments *Scala* type objects, the programmer must use *Scala* type objects too, which is not very convenient. It requires the programmer to have some knowledge of *Scala* which is not necessarily the case.

Since *Java* is one of the most used languages today, *CPChecker* has been made totally compatible with *Java* code. We make sure that a *Java* programmer is able to use only his favorite programming language to test an algorithm with *CPChecker*.

[2]The Java Virtual Machine is the machine that is responsible of running programs written into *Java* bytecode.

For this, in the filtering and pruning classes of *CPChecker*, we defined a constructor overload taking as argument a *Java* checker function :

```
def this(jChecker: Function[Array[Integer], java.lang.Boolean]) =
  this(checkerToScalaFunction(jChecker))
```

where `checkerToScalaFunction` is a function transforming the *Java* `checker` function into a *Scala* function. Thanks to this, the user can create an object of type `ArcFiltering` or another filtering with a *Java* `checker` function passed to the constructor.

But this is not the only place where the knowledge of *Scala* is required. To extend the `Filter` abstract class, a *Scala* function must be implemented. A programmer willing to test his filtering must define `Filter` objects. So, we created an abstract *Java* class `JFilter` that a *Java* programmer can extend to create a `Filter` object using only *Java* code. Here is the code of such a class :

```
abstract class JFilter extends Filter {
  override final def filter(variables: Array[Set[Int]]): Array[Set[Int]] =
    filterJava(variables)

  def filterJava(variables: Array[java.util.Set[Integer]]) :
    Array[java.util.Set[Integer]]
```

The abstract function that must be implemented by the programmer is now the `filterJava` function which takes in arguments a *Java* function.

3.2.8 Examples

Now that everything has been explained about the static testing, let's go over some examples. We realized examples for testing the filtering of some constraints implemented by three solvers : ***OscaR***, ***Choco*** and ***JacoP***. All of them are present in the Github repository containing the source code of *CPChecker* [3]. The usefulness of CPChecker has been proved by having tested the ***OscaR***'s arc consistent filtering for the *Global Cardinality Constraint*. It showed that the filtering is not arc consistent for all variables (count variables).

Let's explain some of these concrete examples over one *Scala* solver (***OscaR***) and one *Java* solver (***Choco***). Since *CPChecker* is written in *Scala*, we will first go through an example for ***OscaR*** before going through the same example but for ***Choco***.

Testing the AllDifferent Constraint

OscaR The first examples will show how one can statically test that the filtering of the *AllDifferent* constraint is arc consistent using *CPChecker*. Let's first show the whole code before going through all the details. As said before, we begin with the ***OscaR*** test.

```
import checker.CPChecker._
import oscar.{...}

object AllDifferentACTest extends App {

  val trustedAllDifferentAC: Filter =
    new ArcFiltering(Checkers.allDifferent())
  val testedAllDifferentAC: Filter = new Filter {
    override def filter(variables: Array[Set[Int]]): Array[Set[Int]] = {
```

[3] `https://github.com/vrombouts/Generic-checker-for-CP-Solver-s-constraints`

```
      filteringAllDifferentAC(variables)
    }
  }
  testArguments.setRangeForAll(-5, 5)
  stats.setFolderName("allDifferentAC")
  check(trustedAllDifferentAC, testedAllDifferentAC)

  def filteringAllDifferentAC(vars: Array[Set[Int]]): Array[Set[Int]] = {
    implicit val testSolver: CPSolver = CPSolver(CPPropagStrength.Strong)
    val variables = vars.map(x => CPIntVar(x))
    val ad = new AllDiffAC(variables)
    try {
      testSolver.post(ad)
    } catch {
      case _: Inconsistency => throw new NoSolutionException
    }
    variables.map(x => x.toArray.toSet)
  }
}
```

This example is not that big. Let's go over it. It begins with the creation of the filtering algorithms that will be compared. For the trusted filtering algorithm, the implementation of *CPChecker* is used. Since the tested filtering should be arc consistent, the trusted filtering has been created as an `ArcFiltering` object. For reminder, this object needs a checker function (see section 3.2.3). For the *AllDifferent* constraint, a checker function has already been implemented in the `Checkers` object. Hence, the trusted algorithm can be created in a single line of code.

The tested filtering algorithm is from ***OscaR***. Its filtering algorithm is implemented in the `filteringAllDifferentAC`. Therefore, to create the `Filter` object, its `filter` function can simply be implemented by calling `filteringAllDifferentAC`.

After that, the default implicit parameters are modified to suit the *AllDifferent* constraint. The test arguments (section 3.1.2) have the range of the variables reduced to augment the number of tests where the domains have some values taken off by the filtering.
The statistics (section 3.2.6) have a new folder to print the results.

Then, there is simply a call to the `check` function of the *CPChecker* object (section 3.2.1). This is this line of code that launches the testing process. Since the test arguments and the statistics are implicit, it only needs the trusted and tested filtering algorithms as parameters.

Let's now look at the `filteringAllDifferentAC` function. This one is more complex. It is the ***OscaR*** arc consistent filtering function for the *AllDifferent* constraint. This function receives an array of domains and must return those domains filtered when they are arc consistent for the *AllDifferent* constraint. Let's remind the function:

```
def filteringAllDifferentAC(vars: Array[Set[Int]]): Array[Set[Int]] = {
  implicit val testSolver: CPSolver = CPSolver(CPPropagStrength.Strong)
  val variables = vars.map(x => CPIntVar(x))
  val ad = new AllDiffAC(variables)
  try {
    testSolver.post(ad)
  } catch {
    case _: Inconsistency => throw new NoSolutionException
  }
  variables.map(x => x.toArray.toSet)
}
```

In this function, the variables received in argument are filtered by the ***OscaR*** solver. At the second line, the ***OscaR*** solver is instantiated as a `CPSolver` named `testSolver`. Once the solver has been instantiated, the constraint will be created. To create the constraint, the variables must first be transformed into the `CPIntVar` object of ***OscaR***. Then, with those `CPIntVar`, the *AllDifferent* constraint can be created. It is done at the fourth line. With the constraint and the solver created, the variables can be filtered. This is done by *'post'*ing the constraint to the solver. This operation is in a try-catch because the solver throws an exception if it filters the domains until one is empty. Once the variables' domains have been filtered until arc consistency, the last step is to re-transform the `CPIntVar` into the simple format of Array of Sets and return them.

This completes the test. When the function `check` will be called, it will generate random domains with its test arguments. Then it will pass those domains to the `trustedAllDifferentAC` and `testedAllDifferentAC`'s `filter` function and compare their return value.

Choco Since *CPChecker* is a tool programmed in *Scala*, it is more intuitive in that language. But *CPChecker* can also be used to test filtering algorithms implemented in *Java*. ***Choco*** is a solver written in *Java*. Let's see how *CPChecker* can be used to test an *AllDifferent* arc consistent filtering written in *Java*.

```
1  public class AllDifferentACTest {
2
3   public static void main(String[] args) {
4     class MyFilter extends JFilter {
5       public Set<Integer>[] filterJava(Set<Integer>[] domains) {
6         return filteringAC(domains);
7       }
8     }
9     Filter trustedAllDifferentAC = new ArcFiltering(Checkers.allDifferent())
      ;
10    Filter testedAllDifferentAC = new MyFilter();
11    TestArgs testArguments = new TestArgs();
12    testArguments.setRangeForAll(-5, 5);
13    Statistics stats = new Statistics("allDifferentAC");
14    CPChecker.check(trustedAllDifferentAC, testedAllDifferentAC, generator,
      stats);
15  }
16
17   public static Set<Integer>[] filteringAllDifferentAC(Set<Integer>[]
      variables) {
18     Model model = new Model("allDifferent problem");
19     IntVar[] x = new IntVar[variables.length];
20     for (int i = 0; i < variables.length; i++) {
21       int[] b = variables[i].stream().
22         mapToInt(Number::intValue).toArray();
23       x[i] = model.intVar("" + i, b);
24     }
25     PropAllDiffAC cstr = new PropAllDiffAC(x);
26     try {
27       cstr.propagate(0);
28     } catch (Exception e) {
29       throw new NoSolutionException("No solution");
30     }
31     return transform(x);
32   }
33
34
```

```
35    /*
36     * returns the domains in the type Set<Integer> from the choco
37     * domains type IntVar
38     */
39    private static Set<Integer>[] transform(IntVar[] input) {...}
40  }
```

This ***Choco*** example is similar to the ***OscaR*** one. All the important parts are the same: a call to the `check` function in the main, the `ArcFiltering` object and the `filteringAllDifferentAC` function.

Let's go through the main function to spot the differences between the *Scala* and *Java* codes. In *Java*, the main can be separated into three parts.

First, the trusted and tested filtering algorithms are instantiated as `Filter` objects. The trusted algorithm is absolutely not different from the one in the *Scala* example. It is an `ArcFiltering` object which takes the checker function of the `Checkers` object representing the *AllDifferent* constraint.
The tested filtering algorithm changes from the *Scala* example. Before, the filtering algorithm was an unnamed class extending the `Filter` class. In *Java*, the extended class is the `JFilter` class (see section 3.2.7). Therefore, the function overriden here is `filterJava`. Except this little change, the creation of the filtering algorithm as a `Filter` object in *Java* is the same as the one in *Scala*.

Second, a `TestArgs` object is created to set the parameters of the tests. As for the *Scala* example, the range of the variables is reduced to be between -5 and 5.
After the parameters for the tests, the statistics must be instantiated. The constructor of the `Statistics` object takes in argument the folder name, hence the `setFolderName` function is not called in this example.

The last part of the main function is the call of the `check` function of the `CPChecker` object. Since *Java* does not support implicit variables, the implicit parameters must be explicitly given to the `check` function. This is why there are four arguments in this *Java* example while there are only two arguments in the *Scala* one.

The `filteringAllDifferentAC` function behave as the *Scala* function written in the previous example but it uses the ***Choco*** solver instead of the ***OscaR*** one. Since this example is coded in *Java*, the function returns and takes as argument a `Set<Integer>[]` object to represents the domains. Let's take a closer look at this function.

```
1  public static Set<Integer>[] filteringAllDifferentAC(Set<Integer>[]
       variables) {
2    Model model = new Model("allDifferent problem");
3    IntVar[] x = new IntVar[variables.length];
4    for (int i = 0; i < variables.length; i++) {
5      int[] b = variables[i].stream().
6        mapToInt(Number::intValue).toArray();
7      x[i] = model.intVar("" + i, b);
8    }
9    PropAllDiffAC cstr = new PropAllDiffAC(x);
10   try {
11     cstr.propagate(0);
12   } catch (Exception e) {
13     throw new NoSolutionException("No solution");
14   }
```

```
15   return transform(x);
16 }
```

First, the solver must be instantiated. In ***Choco***, they use models. So, a *Model* object is instantiated instead.
Second, the variables are transformed from their generic format to the ***Choco*** one (*IntVar* object). This transformation is done at the lines 3 to 8.
Third, the *AllDifferent* constraint over the ***Choco*** variables is instantiated at the line 9.
Then the constraint is propagated to reach the fix-point. This is done in a try-catch because ***Choco*** as ***OscaR*** returns an exception if the variables have empty domains before reaching the fix-point.
Finally, the filtered variables are returned. The `transform` function simply transforms the ***Choco*** variables back to the generic format of array of sets of integers.

Testing the Circuit Constraint

This example tests the *Circuit* constraint (for a reminder of its definition, see section 2.2.5). It shows the usage of *CPChecker*'s `stronger` function. The *Circuit* filtering algorithms for ***OscaR*** and ***Choco*** do not reach the arc consistency. This test will call the `stronger` function to compare the ***OscaR*** and ***Choco*** with an arc consistent bug-free filtering. By using an arc consistent filtering, it will check that the tested implementation does not filter values belonging to a solution (since being stronger than an arc consistent algorithm means removing solutions).

OscaR Like before, the first tested filtering algorithm is from ***OscaR***. Since the tested filtering algorithm will be compared with an arc consistent filtering, the `ArcFiltering` of *CPChecker* will be used. The difference from the *AllDifferent* constraint is that the checker function is implemented in the example and does not come from the `Checkers` object.

Here is the example of ***OscaR*** for the *Circuit* constraint.

```
1  import checker.CPChecker._
2  import oscar.{...}
3
4  object CircuitTest extends App {
5
6    val trusted: Filter = new ArcFiltering(checkerCircuit _)
7    val tested: Filter = new Filter {
8      def filter(variables: Array[Set[Int]]): Array[Set[Int]] =
       filteringCircuit(variables)
9    }
10   testArguments.setRangeForAll((0, 4))
11   testArguments.setDensityForAll(0.8)
12   stronger(trusted, tested)
13
14
15   private def filteringCircuit(vars: Array[Set[Int]]): Array[Set[Int]] = {
16     implicit val solver: CPSolver = CPSolver()
17     val currentVars: Array[CPIntVar] = vars.map(x => CPIntVar(x))
18     val ad = circuit(currentVars)
19     try {
20       solver.post(ad)
21     } catch {
22       case _: Inconsistency => throw new NoSolutionException
23     }
```

```
    currentVars.map(x => x.toArray.toSet)
  }

  def CheckerCircuit(sol: Array[Int]): Boolean = {
    val visited: Array[Boolean] = Array.fill(sol.length)(false)

    def visit(index: Int, acc: Int): Boolean = {
      if (!sol.indices.contains(index) || visited(sol(index)))
        return false
      if (acc == sol.length)
        return sol(index) == 0
      visited(sol(index)) = true
      internal(sol(index), acc + 1)
    }

    visit(sol(0), 2)
  }
}
```

It first begins with the instantiations of the `Filter` object. They are similar to the `Filter` objects of the *AllDifferent* example. The only important difference is that the `ArcFiltering` object does not take a checker function from the `Checkers` object. It uses `checkerCircuit` function.

After that, the `testArguments` variable that is responsible of the generated domains has been set. For the generated domains, by default there are 5 variables and the *Circuit* constraint imposes that the variables' domains are the indices of the variables. Hence the range as been set to $[0, 4]$. Since the range has been reduced, the density has been set to 0.8 to avoid too small domains.

As said before, the check will be done by the `stronger` function. To have more details about this function, go to section 3.2.2.

Let's now see the two functions `filteringCircuit` and `checkerCircuit` implemented to test the *Circuit* constraint.

The `filteringCircuit` function receives the domains as argument and returns the domains filtered by the ***OscaR*** filtering algorithm of the *Circuit* constraint. It behaves in the same way as the `filteringAllDifferentAC` of the previous ***OcsaR*** example. The only difference is that the constraint created is the *Circuit* instead of the *AllDifferent* one.

The `checkerCircuit` serves as a checker function. Hence it takes an instantiation as argument and returns true if the instantiation respects the *Circuit* constraint. Previously, all the checker functions were taken from the `Checkers` object of *CPChecker*. Here, the user implemented its own checker.
Let's see in more details its implementation.

```
def CheckerCircuit(sol: Array[Int]): Boolean = {
  val visited: Array[Boolean] = Array.fill(sol.length)(false)

  def visit(index: Int, acc: Int): Boolean = {
    if (!sol.indices.contains(index) || visited(index))
      return false
    if (acc == sol.length)
      return index == 0
    visited(index) = true
    internal(sol(index), acc + 1)
  }
```

```
12
13    visit(sol(0), 1)
14  }
```

The logic of this checker is simple. A solution for the *Circuit* constraint represents a circuit which starts from a certain index and returns back to this index after having gone through all the other indices. From this, two important properties can be observed. First, while going through a circuit, it is not possible to pass by an index twice before finishing a round. Second, if there are n indices, the same index is separated by $n - 1$ indices. With those properties, a checker function can be created.

This checker function simply goes through the whole circuit once. It starts at the node 0 and must finish at node 0 to be true. The `visit` function visits an index. It possesses an accumulator which tells how many indices have already been visited. It can also access the boolean array `visited` which tells if a variable (index) has already been visited or not. At the start of the visits, the index 0 is not set as visited because it should be the last node visited for the circuit. Since it is not set as visited, the first *if* condition catches correctly wrong instantiation that have become stuck in a loop. It also catches if the index should be a part of the circuit. The second *if* condition verifies if the circuit is finished thanks to the accumulator. If the circuit is finished (the accumulator is equal to the length of the solution), the circuit is correct if the circuit is at the starting index (0).

With this `checkerCircuit` function, the trusted filtering algorithm can be created. Notice that the *CPChecker*'s bug-free filtering algorithm will be based on the checker function. So, when the user provides its own implementation of a checker function, it is his responsibility to provide a bug-free checker. If the user makes an error in the checker function, *CPChecker*'s behavior might not be the expected one.

Choco After the example in *Scala*, comes the one in *Java*. Therefore, the ***Choco*** solver also had its *Circuit* constraint tested. As for the ***OscaR*** example, the checker function will also be implemented in the example. Let's have a first look at the implementation.

```
1  public class CircuitTest {
2
3    public static void main(String[] args) {
4      class MyFilter extends JFilter {
5        public Set<Integer>[] filterJava(Set<Integer>[] variables) {
6          return filteringCircuit(variables);
7        }
8      }
9      Filter trusted = new ArcFiltering(checkerCircuit());
10     Filter tested = new MyFilter();
11     TestArgs parameters = new TestArgs();
12     parameters.setRangeForAll(0, 4);
13     parameters.setDensityForAll(0.8);
14     Statistics stats = new Statistics("");
15     CPChecker.stronger(trusted, tested, parameters, stats);
16   }
17
18   private static Set<Integer>[] filteringCircuit(Set<Integer>[] variables)
19   {
20     Model model = new Model("Testing choco's circuitSCC filtering");
21     IntVar[] currentVars = new IntVar[variables.length];
22     for (int i = 0; i < variables.length; i++) {
23       int[] b = variables[i].stream().mapToInt(Number::intValue).toArray();
24       currentVars[i] = model.intVar("" + i, b);
```

```
    }
    Constraint ctr = new Constraint(ConstraintsName.CIRCUIT, new
      PropCircuitSCC(currentVars, 0, CircuitConf.ALL));
    model.post(ctr);
    try {
      model.getSolver().propagate();
    } catch (Exception e) {
      throw new NoSolutionException("No solution");
    }
    return transform(currentVars);
  }

  private static boolean visit(Integer[] variables, int index, int acc,
  boolean[] isVisited) {
    if (index < 0 || index >= variables.length) return false;
    if (isVisited[index]) return false;
    if (acc == variables.length) return index == 0;
    isVisited[index] = true;
    return visit(variables, variables[index], acc + 1, isVisited);
  }

  private static Function<Integer[], Boolean> checkerCircuit() {
    return variables -> {
      boolean[] isVisited = new boolean[variables.length];
      return visit(variables, variables[0], 1, isVisited);
    };
  }

  private static Set<Integer>[] transform(IntVar[] input) {...}
}
```

This example possesses three functions: `filteringCircuit`, `checkerCircuit` and the main function.
The main function can be separated into three parts: the instantiation of the `Filter` objects, the setting of the implicit parameters and the call to the `stronger` function.

Since this example is coded in *Java*, the `JFilter` is extended to create the tested `Filter` object. The trusted filtering algorithm does not change from the *Scala* example. It is still an `ArcFiltering` object with the checker function obtained thanks to the `checkerCircuit` function.

Since *Java* does not support the implicit parameters, the `TestArgs` and `Statistics` objects are created in the example. For this same reason, the `stronger` function takes 4 arguments. Namely the trusted filtering algorithm, the tested filtering algorithm, the test arguments (`parameters`) and the statistics.

The filtering algorithm of ***Choco*** goes like that:

```
private static Set<Integer>[] filteringCircuit(Set<Integer>[] variables)
{
  Model model = new Model("Testing choco's circuitSCC filtering");
  IntVar[] currentVars = new IntVar[variables.length];
  for (int i = 0; i < variables.length; i++) {
    int[] b = variables[i].stream().mapToInt(Number::intValue).toArray();
    currentVars[i] = model.intVar("" + i, b);
  }
  Constraint ctr = new Constraint(ConstraintsName.CIRCUIT, new
    PropCircuitSCC(currentVars, 0, CircuitConf.ALL));
```

```
11    model.post(ctr);
12    try {
13      model.getSolver().propagate();
14    } catch (Exception e) {
15      throw new NoSolutionException("No solution");
16    }
17    return transform(currentVars);
18  }
```

First a model is created. Then the given domains are transformed into ***Choco***'s variable's type. After that, the *Circuit* constraint of ***Choco*** is created. The filtering algorithm of ***Choco*** for the *Circuit constraint* is based on the principle of Strongly Connected Components. Since it is not the objective of this thesis, this algorithm will not be explained.

Once the constraint is created, it is posted to the model. As ***Choco*** is different from ***OscaR***, the `post` function does not automatically reach the fix-point. Therefore, a call to the ***Choco***'s `propagate` function must be made to reach it. Once it is done, the filtered domains are retrieved and returned in the generic format. For reminder, the `transform` function transforms the ***Choco***'s variables into the generic format of array of sets of integers.

Let's now look at the most important part of this example. Before, everything was extremely similar to already seen examples. But the `checkerCircuit` function has an important particularity that a user must know to use *CPChecker* in *Java*. This particularity appears because *Java* does not allow functions to be passed as arguments. Since *Java* does not allow functions as arguments, how can the `ArcFiltering` object be created in the main function? It can be done thanks to the functionality provided by *Java8*. Let's see the code of the checker function in more details.

```
1  private static Function<Integer[], Boolean> checkerCircuit() {
2    return variables -> {
3      boolean[] isVisited = new boolean[variables.length];
4      return visit(variables, variables[0], 1, isVisited);
5    };
6  }
7  private static boolean visit(Integer[] variables, int index, int acc,
8    boolean[] isVisited) {
9    if (index < 0 || index >= variables.length) return false;
10   if (isVisited[index]) return false;
11   if (acc == variables.length) return index == 0;
12   isVisited[index] = true;
13   return visit(variables, variables[index], acc + 1, isVisited);
14 }
```

A checker function should normally be a function which takes an array of integers and return a boolean value. Here, the `checkerCircuit` takes no argument and returns a `Function` object. In fact, the real function passed to the `ArcFiltering` is not the `checkerCircuit` but the `Function` object it returns. There are two ways to create a Function object.

First, it is possible to implement a class which extends `Function` and implements its `apply` function.

The second way is to use lambda expressions. Here, a lambda expression has been used. Just by writing the lines 2 and 5, it is possible to consider the inside of the brackets as the body of the returned function. This returned function takes the `variables` as argument.

The behavior of the checker function represented by the `Function` object will not be explained

since it behaves exactly in the same way as the `checkerCircuit` function of the ***OscaR*** example for the *Circuit* constraint.

3.3 Testing Incremental Filtering Algorithms

In this section, we will develop how to incrementally test constraints using *CPChecker*. The incremental testing allows the developer to test the trailing that is often used for the backtracking procedure during the search. Such a procedure relies on domains storage and restoration, implemented by push and pop operations. Any implementation that relies on the trailing mechanisms could be tested by using this part.

In order to test that the domains' storage and restoration is well performed, we will generate random domains as well as random branch operations. Afterwards, we will explain how these branch operations are generated.

In this part, we will check that the tested filtering of the domains for a constraint is correctly performed and also that during the search, while doing domain's restrictions and trailing operations (push, pop), the filtering of the constraint continues to work correctly.
To do so, we will first check that the tested filtering of the initial domains until reaching fix-point is correctly performed compared to the trusted one thanks to their *setup* function. This function should filter domains until reaching fix point (as in the static part). The only difference with the static part here is that, if the tested filtering involves a solver, the solver's initialization must be performed in this function too.

Then, given these filtered domains, we will test the application of the filtering when doing restrictions of domains or push/pop operations thanks to their *branch-and-filter* function. Given a branch operation (domain restriction, push, pop) and the current domains, this function should return the reduced domains after having performed the operation and filtered the domains until reaching a new fix-point. The process of incrementally testing a filtering algorithm can be described by these steps:

1. Domains generation : as in the static part, we generate some random domains that will be used to test the user's implementation. These domains are configurable by the user (see section 3.1.2).

2. Application of the tested *setup* function.

3. Application of the trusted *setup* function. As in the static part, in the case where the trusted filtering is the *CPChecker*'s filtering, we will need a *checker* function to perform this step. It is necessary for the *CPChecker* to know which constraint is being filtered.

4. Comparing the filtered domains generated by the tested filtering (in step 2) and the ones generated by the trusted filtering (in step 3). If they are not the same, we will stop the check process and display an error message. Otherwise, we will go to step 4.

5. Creation of branches. In this step, we will generate random branches in order to test the behavior of the tested implementation when doing either a restriction of a domain, a push or a pop operation.

6. Application of the tested *branch-and-filter* function.

7. Application of the trusted *branch-and-filter* function.

8. Comparison of the filtered domains in 6 and 7. If the comparison reveals the tested filtering is not correct, an error message is displayed and the test stops. Otherwise, it goes back to

step 5 to branch again until reaching a certain number of branch operations. The branching procedure will be explained in section 3.3.4.

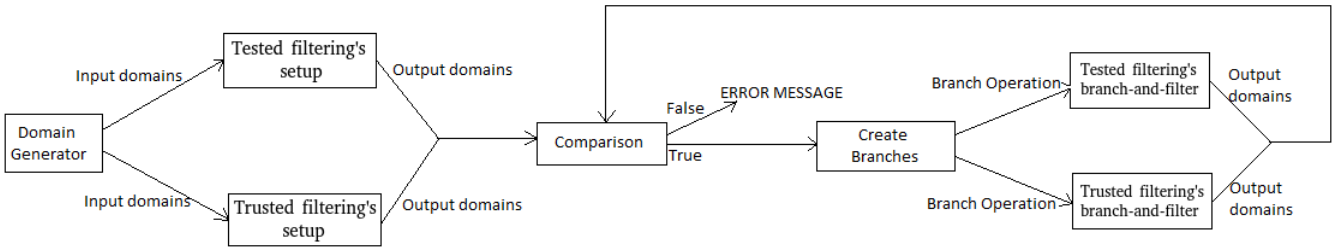


Figure 3.2: Schematic representation of the incremental testing of one generation of domains.

The incremental testing process has been represented on this figure (Fig. 3.2). This figure represents the whole process of incrementally testing the user's implementation over some domains. This process is repeated by default 100 times with different test instances (different domains are generated).

Let's explain in more details the filtering functions that must be implemented by the user.

Setup function As explained previously, in terms of input/output, the *setup* function is similar to the filtering function of the static part. So, given the input domains, it has to return the filtered domains when reaching the fix point. But it is also the function that is responsible of the solver's initialization. By solver's initialization, we mean the solver's declaration and the addition of the constraint to be tested in the solver's constraint store[4]. Later, while doing domain restriction, the constraint representing the domain restriction will be added to the same solver's constraint store and the push/pop operations have to be performed over this solver too. Here is the signature of such a function :

```
1 def setup(variables: Array[Set[Int]]): Array[Set[Int]]
```

Branch-and-filter function The *branch-and-filter* function takes in argument a branch operation. It has to consider the previously computed domains and to apply the branch operation over these domains. After that, it has to propagate the constraint until fix-point and return these filtered domains. Here is the signature of such a function :

```
1 def branchAndFilter(branching: BranchOp): Array[Set[Int]]
```

Here are the formal specifications of this function :

- pre-conditions : `branching` is a valid branch operation : either a push, pop or a domain's restriction. A domain's restriction operation will never involve a variable having an empty domain.
- post-conditions : returns the filtered domains reaching fix-point with respect to the constraint after the application of the `branching` over the previously computed domains. If during the propagation it founds there is no solution given the constraint, it can either return empty domains or throw a `NoSolutionException`.

[4]The solver's constraint store refers to the set of active constraints that the solver must consider during the search.

3.3.1 Equality of Filtering Algorithms

In order to check that two filtering algorithms are equal, the user can call a `check` function as in the static part. This function takes arguments that are different from the ones of the static `check` function but performs the comparison in a similar way.

```
def check(bugFreeFiltering: FilterWithState,
          testedFiltering: FilterWithState)
         (implicit testArguments: TestArgs, stats: Statistics): Boolean = {
  var result = true
  forAll(testArguments.gen) { x =>
    if ((x.length < testArguments.getNbVars) || checkEmpty(x)) true
    else if (!checkConstraint(x.toArray, bugFreeFiltering, testedFiltering,
                              comparisonCheck(_, _, stats), testArguments))
    {
      result = false
      false
    }
    else true
  }.check(testArguments.getTestParameters)
  stats.setGenerator(testArguments)
  stats.print(true)
  result
}
```

The code is similar to the one of the `check` function of the static part. The major difference lies in the argument types which are `FilterWithState` objects. To understand, let's explain the `FilterWithState` class. This class is an abstract class containing two abstract functions : `setup` and `branchAndFilter`. The creation of a `FilterWithState` object involves the implementation of these two functions. Here is the code of this abstract class :

```
abstract class FilterWithState {
  def setup(variables: Array[Set[Int]]): Array[Set[Int]]

  def branchAndFilter(branching: BranchOp): Array[Set[Int]]
}
```

With this, the `check` function will test that the filtered domains returned at each call to the `setup` and `branchAndFilter` functions are the same for the tested and trusted `FilterWithState` objects.

3.3.2 Comparing Strength of Filtering Algorithms

To compare the strength of two filtering algorithms, the user can call the `stronger` function taking in arguments two `FilterWithState` objects. The function is very similar to the `stronger` function of the static part. Here is the signature of this function :

```
def stronger(strongerFiltering: FilterWithState,
             weakerFiltering: FilterWithState)
            (implicit testArguments: TestArgs, stats: Statistics): Boolean
```

This function takes as first argument the `FilterWithState` object which should filter the domains the most and as second argument the `FilterWithState` which should filter less. Testing that one `FilterWithState` object filters more than another is quite simple. It will check that after the application of the `setup` function, the domains that have been filtered by the `strongerFiltering` are subsets of the other filtered domains. Then, it checks that this is always the case at each call to a `branchAndFilter` function. Note that it is not a strict stronger relation : if the two `FilterWithState` objects filter the domains in the same way, the `stronger` function

will consider it passes the test. However, in the case where the second filtering algorithm filters more domains than the first one, the test fails.

3.3.3 Testing Different Propagation Types

CPChecker provides implementations reaching arc, bound and range consistencies. It also provides an implementation for simulating the trailing mechanisms (push, pop). The class `IncrementalFiltering` contains the implementations of the `setup` and `branchAndFilter` functions implementing the filtering process to respond to branch operations.

```
1 class IncrementalFiltering(filter: Filter) extends FilterWithState
```

To allow the user to choose the kind of consistency reached by the filtering, the constructor of the `IncrementalFiltering` class takes in argument a `Filter` object. The user can then choose between the *CPChecker*'s filtering implementations : `ArcFiltering`, `BoundZFiltering`, `BoundDFiltering` or `RangeFiltering`, as well as an `HybridFiltering` or their pruning counterparts. He can even choose any other trusted static filtering algorithm implemented as a `Filter` object to be passed to the constructor if he wishes so. Note that it lets to the user the freedom to combine one of its static filtering algorithms with our implementation involving the trailing mechanisms if he wants to. The way how these trailing mechanisms are implemented will be explained later in section 3.3.4.

3.3.4 Generation of Branch Operations

With the `FilterWithState` class, the incremental testing is done by the `branchAndFilter` function after the `setup` function has been called. The objective is to test the correctness of the filtering after a trail operation or going through a child by restricting one of the domains. Therefore, the filtering algorithms are brought to create a pseudo-search tree by following the branch operations given to the filtering function. Those are the three types of branching operations *CPChecker* can generate:

1. *Push* : The current state of the solver must be *pushed* on its trail.
2. *Pop* : The solver must restore its last state *pushed* on its trail by using the *pop* operation of the trail.
3. *RestrictDomain* : This branching operation gives a constraint over a single variable (therefore, this constraint restricts a domain). The solver must add this new constraint and reach the new fix-point. The given constraint is very simple and consists in applying a simple operator with its constant such as '< 3' or '$= 5$'. Let $x_1 = \{0, 1, 2\}, x_2 = \{1, 3, 5\}, x_3 = \{4, 5, 6, 8\}$ be the variables. A domain restriction operation can be : $x_1 > 0$, $x_1 < 2$, $x_2 = 3$, $x_3 \neq 6$,...

All of those branching operations are represented by their own class extending the `BranchOp` class which represents a branching operation in general.

The constraints represented by *RestrictDomain*'s are randomly chosen. Nevertheless, we pay particularly attention to only consider interesting operations. For example, looking again to the example above, we will never create from this an operation such that $x_1 < 0$ which will of course always lead to no solution, whatever the constraint being tested is.

Dives

To generate a search tree, *CPChecker* performs dives. The principles lying under dives are quite simple. It consists of going from one node to another until reaching a leaf. Then, it backtracks a

random number of times in the tree. The node that it reaches after these backtracks will then serve as the first node of the next dive and so on. Here is a schematic representation of the diving process :

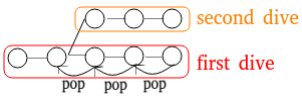


Figure 3.3: Schematic representation of two successive dives

Here, each node symbolizes the domains variables. Going from one node to another consists in doing a domain restriction. Before each domain restriction, the current domains are pushed on a trail. The backtracks are realized by popping the trail. You can see here in details the algorithm we use to perform dives :

Algorithm 1: Algorithm performing dives

```
Dives (root, trail, nbDives)
    inputs : root, trail, nbDives
    dives ← 0
    currentDomains ← root
    while dives<nbDives do
        while !currentDomains.isLeaf do
            trail.push(currentDomains)
            restriction ← new RandomRestrictDomain(currentDomains)
            currentDomains ← branchAndFilter(currentDomains, restriction)
        dives ← dives + 1
        for i ← 1 to Random(1,trail.size-1) do
            trail.pop()
```

Performing a dive from a given node works as follow: if the state of the node is not pushed, push it in the trail. Otherwise, go to one of its children by restricting the domain of one variable chosen randomly. Then, apply the trusted `branchAndFilter` function over the restricted domains in order to have the restricted domains filtered by the trusted algorithm. Repeat these steps until a leaf has been reached. For reminder, a node is a leaf if all variables are fixed (with size 1 domains) or if at least one of the variables has an empty domain. All dives performed by *CPChecker* do not start at the root of the search tree. When a dive has been performed, the node to start the next dive is found by popping the trail a random number of times. This random number is between 1 and the number of states currently pushed in the trail. This range is used to avoid a pop operation when the trail is empty and to avoid staying in the leaf node.

Here is the representation of the diving process over an example :

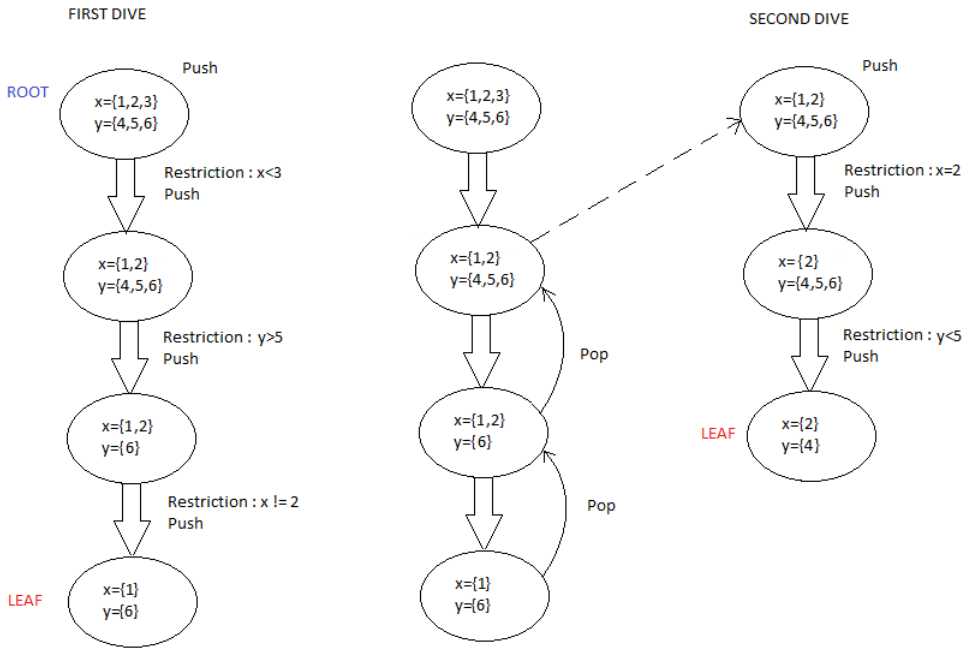


Figure 3.4: Example of dives

When two filtering algorithms are compared, the number of dives by test is fixed. By default, `TestArgs` sets the number of dives to 10. This number can be modified by changing the value of the *nbDives* parameter to the wished number of dives. Nevertheless, some tests do not perform any dive. Indeed, there are some tests where all variables are already fixed or the problem has been found unsatisfiable at the root node (after having called the *setup* function). In those tests, there is no domain that can be restricted. Therefore, in these cases, we do not perform any dive.

Dealing with Branch Operations

Let's detail how the branching operations are dealt with in the `IncrementalFiltering` class. Since there are push/pop operations, we decided to record the states into a list which will behave as a stack. So, in the `IncrementalFiltering` class, there is a variable named `trailStack`. This variable is of type `List[Array[Set[Int]]]` in order to record all variables states. It represents the trail mechanisms. Since the filtering algorithm is a static filtering algorithm (the one given in argument to the `IncrementalFiltering`), the trail mechanisms can simply be reduced to only the `trailStack` variable.

From this, let's detail the behaviour of the *CPChecker* when dealing with each operation :

Push When dealing with a push operation, the current domains are simply recorded to be restored later. So, they are pushed on the `trailStack`. Here is the line of code to do it :

```
1 def push(currentDomains: Array[Set[Int]]): Array[Set[Int]] = {
2   trailStack = currentDomains :: trailStack
3   currentDomains
4 }
```

The domains after the push operation have not been modified. The only thing to do is to keep them in memory for later uses. They are kept as the new head of the list. At the end, the `currentDomains` are returned.

Pop When dealing with a pop operation, the current domains are simply restored to their previous recorded state. So, the domains are popped from the `trailStack`. Here are the lines of code to do it :

```
def pop(currentDomains: Array[Set[Int]]): Array[Set[Int]] = {
  if (trailStack.nonEmpty)
    val vars = trailStack.head
    trailStack = trailStack.tail
    vars
  else
    currentDomains
}
```

The `if` condition has been made in order to prevent errors. It requires that the `trailStack` is not empty before performing a Pop operation over it. If it is empty, the function simply returns the `currentDomains`. Otherwise, it returns the popped domains. Notice that the trailStack is reset without its head after the pop operation.

Domain Restriction The last operation is the domain restriction. This operation has nothing to do with the stack but simply reduces one variable's domain. So, it relies on three parameters : an operation, an index and a constant representing the restriction to apply. These three parameters are simply referred to as `op`, `index` and `constant`. Here is the code of the application of the domain reduction :

```
def applyRestriction: Array[Set[Int]] = {
  domains(index) = domains(index).filter(value => Op.respectOp(op, value,
    constant))
  domains
}
```

In this function, the restricted domain is filtered thanks to the *Scala* `filter` function. The `respectOp` function takes as argument an operation and two integers. It looks if the operation is respected between the two integers. Here, the first integer refers to a value of the restricted domain. Hence, if the operation is not respected, the value is removed from the domain. This function then returns the domains.

With those functions, the `setup` and `branchAndFilter` functions of the `IncrementalFiltering` are easy to understand:

```
override def setup(variables: Array[Set[Int]]): Array[Set[Int]] = {
  trailStack = List()
  filter.filter(variables)
}

override def branchAndFilter(branching: BranchOp): Array[Set[Int]] = {
  var restrictDomain: Array[Set[Int]] = Array()
  branching match {
    case _: Push => push(branching.domains)
    case _: Pop => pop(branching.domains)
    case restriction: RestrictDomain =>
      restrictDomain = restriction.applyRestriction
      filter.filter(restrictDomain)
    case _ => branching.domains
  }
}
```

3.3.5 User Report

When calling the `check` or `stronger` function, some output is created in order to give to the user a detailed view of what works well and what goes wrong when comparing the two algorithms passed as arguments. For this, some prints are displayed with the pass/fail outcome, as in the static part. In addition to that, three files are generated in the *out/statistics* repository : *statistics.txt*, *passedTests.txt* and *failedTests.txt*. Here is the display of the *statistics.txt* file generated while calling `check` over two algorithms filtering the sum constraint considering a number of dives of 3 :

```
Here are the statistics of the executed tests :
-----------------------------------------

Number of executed tests : 2965
Number of failed tests : 0

Tested and trusted algorithms filters the same : 2965
Tested algorithm filters less than the trusted : 0
Tested algorithm filters more than the trusted (stronger) : 0
Trusted algorithm and tested one filters differently
(without one stronger than the other) : 0

Trusted algorithm filters no value : 438

Ratio of filtering until no solution found (tested algo/ trusted algo) : 108/108
Ratio of filtering until a single solution (tested algo/ trusted algo) : 192/192

The average depth reached by the dives is 3
The maximum depth reached is 10

Generator :
        Seed: random
        variable    density range
        0           0.2     (-10,10)
        1           0.2     (-10,10)
        2           0.2     (-10,10)
        3           0.2     (-10,10)
        4           0.2     (-10,10)
```

You can see that, even though we let the parameter representing the number of tests to 100, the number of executed tests is quite big : 2965. It is because of the fact that this number represents the number of comparisons and not the number of test cases. Since each test case performs 3 dives, it makes a lot of comparisons to be performed. Except for that, this file is very similar to the *statistics.txt* file in the static output part (see section 3.2.6).

An extract of the corresponding *passedTests.txt* file looks like :

```
Initial domains   |Trusted filtering |Tested filtering
[-1, -10, -8, -5] |[-1, -10, -8, -5] |[-5, -1, -8, -10]
[-10, -3, 3, 0]   |[-3, 3, 0]        |[0, -3, 3]
[5, -8, 6, -4]    |[5, 6, -4]        |[5, 6, -4]
[0, -7, -1, 6]    |[0, -1, 6]        |[0, -1, 6]
[-1, -4, 1, -10]  |[-1, -4, 1]       |[1, -1, -4]
```

```
TEST PASSED :
Init : [[-1,-10,-8,-5][-10,-3,3,0][5,-8,6,-4][0,-7,-1,6][-1,-4,1,-10]]
Push: [[-1,-10,-8,-5][-3,3,0][5,6,-4][0,-1,6][-1,-4,1]]
Restriction of domains (x_1!=-3): [[-1,-10,-8,-5][3,0][5,6,-4][0,-1,6][-1,-4,1]]
Push: [[-1,-10,-8,-5][3,0][5,6,-4][0,-1,6][-1,-4,1]]
Restriction of domains (x_4>=1): [[-1,-10,-8,-5][3,0][5,6,-4][0,-1,6][1]]
Push: [[-1,-10,-8,-5][3,0][5,6,-4][0,-1,6][1]]
Restriction of domains (x_1>0): [[-1,-10,-8,-5][3][5,6,-4][0,-1,6][1]]
Push: [[-1,-10,-8,-5][3][5,6,-4][0,-1,6][1]]
Restriction of domains (x_0<=-8): [[-10][3][5][6][1]]
Pop: [[-1,-10,-8,-5][3][5,6,-4][0,-1,6][1]]
Pop: [[-1,-10,-8,-5][3,0][5,6,-4][0,-1,6][1]]
Pop: [[-1,-10,-8,-5][3,0][5,6,-4][0,-1,6][-1,-4,1]]
Push: [[-1,-10,-8,-5][3,0][5,6,-4][0,-1,6][-1,-4,1]]
Restriction of domains (x_2=5): [[-1,-10,-8,-5][3,0][5][0,-1,6][-1,-4,1]]
Push: [[-1,-10,-8,-5][3,0][5][0,-1,6][-1,-4,1]]
Restriction of domains (x_3>-1): [[-1,-10,-8,-5][3,0][5][0,6][-1,-4,1]]
Push: [[-1,-10,-8,-5][3,0][5][0,6][-1,-4,1]]
Restriction of domains (x_0<=-5): [[-10,-8,-5][3,0][5][6][-1,-4,1]]
Push: [[-10,-8,-5][3,0][5][6][-1,-4,1]]
Restriction of domains (x_1>=3): [[-10,-8,-5][3][5][6][-1,-4,1]]
Push: [[-10,-8,-5][3][5][6][-1,-4,1]]
Restriction of domains (x_0<-5): [[-10,-8][3][5][6][-1,1]]
Push: [[-10,-8][3][5][6][-1,1]]
Restriction of domains (x_4!=-1): [[-10][3][5][6][1]]
Pop: [[-10,-8][3][5][6][-1,1]]
Pop: [[-10,-8,-5][3][5][6][-1,-4,1]]
Push: [[-10,-8,-5][3][5][6][-1,-4,1]]
Restriction of domains (x_4=-4): [[-5][3][5][6][-4]]
------------------------------
```

This extract first displays the tests concerning the first filtering that is made while executing a test, that is, the filtering reaching fix-point (obtained while calling the `setup` function). Then, the incremental tests are displayed. All the branching stages are described by detailing the kind of operation and the returned domains by the trusted `branchAndFilter` function for this operation.

Let's now have a look to a failed test :

```
TEST FAILED :
Init : [[0][-3,-7,3,7][4,-6,-9,0][-4,-7,1][-9,9,-8,5]]
Push: [[0][-3,-7,3,7][4,-6,-9,0][-4,-7,1][9,5]]
Restriction of domains (x_4>5): [[0][-3,-7,3,7][4,-6,-9,0][-4,-7,1][9]]
Push: [[0][-3,-7,3,7][4,-6,-9,0][-4,-7,1][9]]
Restriction of domains (x_2!=0): [[0][-3,-7,3,7][4,-6,-9][-4,-7,1][9]]
Push: [[0][-3,-7,3,7][4,-6,-9][-4,-7,1][9]]
Restriction of domains (x_2>=-6): [[0][-3,-7,3,7][4,-6][-4,-7,1][9]]
Push: [[0][-3,-7,3,7][4,-6][-4,-7,1][9]]
Restriction of domains (x_1<7): [[0][-3,-7,3][4,-6][-4,-7,1][9]]
Push: [[0][-3,-7,3][4,-6][-4,-7,1][9]]
Restriction of domains (x_1=3) failed:
```

```
After the application of Restriction of domains (x_1=3):
Initial domains |Trusted domains |Tested domains
[0]             |[]              |[0]
[-3, -7, 3]     |[]              |[3]
[4, -6]         |[]              |[-6, 4]
[-4, -7, 1]     |[]              |[1, -4, -7]
[9]             |[]              |[9]

------------------------------
```

The different steps of the test are displayed and then, when the failing occurs, the `initial`, `trusted` and `Tested` domains are displayed in order to allow the user to compare them.

3.3.6 From Scala to Java

As said before, we can't assume that any developer has a good knowledge of *Scala*. Since *Java* is a very used language, we made *CPChecker* callable using only *Java* code. We already defined how this has been achieved for the consistency classes and for the `Filter` class in the static part (section 3.2.7).

For the abstract `FilterWithState` class, as for the `Filter` class with its `JFilter` *Java* counterparts, it has its own `JFilterWithState` class :

```
1  abstract class JFilterWithState extends FilterWithState {
2    override final def setup(variables: Array[Set[Int]]): Array[Set[Int]] =
       setupJava(variables)//implicit conversion
3
4    override final def branchAndFilter(branching: BranchOp): Array[Set[Int]]
       = branchAndFilterJava(branching)//implicit conversion
5
6    def setupJava(variables: Array[java.util.Set[Integer]]):
7      Array[java.util.Set[Integer]]
8
9    def branchAndFilterJava(branching: BranchOp):
10     Array[java.util.Set[Integer]]
11 }
```

So, a *Java* user can decide to extend the `JFilterWithState` class instead of the `FilterWithState` one by specifying the body of the `setupJava` and `branchAndFilterJava` classes taking *Java* arguments.

3.3.7 Examples

Now, everything about the incremental testing of filtering algorithms has been seen. Let's see how it looks in practice.
This section shows two examples of testing incremental filtering algorithms. One is written in *Scala* and uses the ***OscaR*** solver and the second is in *Java* and uses the ***Choco*** solver. Both test their filtering algorithm for the *Sum* constraint. For more details about this constraint, refers to the section 2.2.2. More precisely, the constraint that is considered here is: $\sum x = 15$.

OscaR

The filtering algorithm of ***OscaR*** for the *Sum* constraint is bound($\mathcal{Z}$) consistent. Therefore, the `BoundZFiltering` class will be used to create the trusted filtering algorithm. Let's see the example.

```
object SumBCIncrTest extends App {

  testArguments.setSeed(1000)
  val trusted = new IncrementalFiltering(
    new BoundZFiltering(Checkers.sum(15, "=")))
  val tested = new FilterWithState {
    implicit private var solver: CPSolver = new CPSolver
    private var currentVars: Array[CPIntVar] = _

    override def branchAndFilter(branching: BranchOp): Array[Set[Int]] = {
      branch match {
        case _: Push =>
          solver.pushState()
          currentVars.map(x => x.toArray.toSet)
        case _: Pop =>
          solver.pop()
          currentVars.map(x => x.toArray.toSet)
        case r: RestrictDomain =>
          try {
            val constant = r.constant
            val variable = currentVars(r.index)
            var c: oscar.cp.Constraint = null
            r.op match {
              case "=" => c = new EqCons(variable, constant) // x(i)=c
              case "<" => c = new Le(variable, constant) // x(i)<c
              case ">" => c = new Gr(variable, constant) // x(i)>c
              case "!=" => c = new DiffVal(variable, constant) // x(i)!=c
              case "<=" => c = new LeEq(variable, constant) // x(i)<=c
              case ">=" => c = new GrEq(variable, constant) // x(i)>=c
            }
            solver.post(c)
          } catch {
            case _: oscar.cp.core.NoSolutionException =>
              throw new NoSolutionException
            case _: Inconsistency =>
              throw new NoSolutionException
          }
          currentVars.map(x => x.toArray.toSet)
        case _ => currentVars.map(x => x.toArray.toSet)
      }
    }

    override def setup(variables: Array[Set[Int]]): Array[Set[Int]] = {
        solver = CPSolver()
      currentVars = vars.map(x => CPIntVar(x))
      val ad = sum(currentVars).eq(15)
      try {
        solver.post(ad)
      } catch {
        case _: Inconsistency => throw new NoSolutionException
        case _: oscar.cp.core.NoSolutionException =>
          throw new NoSolutionException
      }
      currentVars.map(x => x.toArray.toSet)
    }
  }

  check(trusted, tested)
```

```
}
```

The first step is the setting of a seed. Like that, the tests will stay the same no matter the number of times the example is run. Then, the filtering algorithms are created. As said before, since the consistency that should be reached here is the bound($\mathcal{Z}$) one, the trusted filtering algorithm is created thanks to the `BoundZFiltering` class. But it is not enough. Here, the filtering algorithm is incrementally tested. Hence a `FilterWithState` object must be used instead of a `Filter` one. To have the `FilterWithState`, the `BoundZFiltering` object is passed as argument to an `IncrementalFiltering` object which extends `FilterWithState`. For reminder, this class takes a static filtering algorithm as argument and represents the static filtering to be comparable in an incremental test.
Since `BoundZFiltering` needs a checker, the one from the `Checkers` object has been used.
The tested filtering algorithm is directly created as a `FilterWithState` and implements the `setup` and `branchAndFilter` functions. Finally, the `check` function is called over the trusted and tested filtering algorithms.

We will now observe the `setup` and `branchAndFilter` functions.

Here is the `setup` function:

```
def setup(vars: Array[Set[Int]]): Array[Set[Int]] = {
  solver = CPSolver()
  currentVars = vars.map(x => CPIntVar(x))
  val ad = sum(currentVars).eq(15)
  try {
    solver.post(ad)
  } catch {
    case _: Inconsistency => throw new NoSolutionException
    case _: oscar.cp.core.NoSolutionException =>
      throw new NoSolutionException
  }
  currentVars.map(x => x.toArray.toSet)
}
```

It is similar to the `filter` functions of the `Filter` objects implemented in the static examples of ***OscaR***. The only differences are the constraint posted and the variables `solver` and `currentVars`. Those variables are not local variables. `solver` and `currentVars` can be accessed outside the `setup` function. It is done to permit the `branchAndFilter` function to access the current state of the model.
Since the `branchAndFilter` function can access the current state of the problem, let's see what it does when it receives a branching operation.

```
def branchAndFilter(branch: BranchOp): Array[Set[Int]] = {
  branch match {
    case _: Push =>
      solver.pushState()
      currentVars.map(x => x.toArray.toSet)
    case _: Pop =>
      solver.pop()
      currentVars.map(x => x.toArray.toSet)
    case r: RestrictDomain =>
      try {
        val variable = currentVars(r.index)
        val constant = r.constant
        var c: oscar.cp.Constraint = null
        r.op match {
          case "=" => c = new EqCons(variable, constant) // x(i)=constant
```

```
16            case "<" => c = new Le(variable, constant) // x(i)<constant
17            case ">" => c = new Gr(variable, constant) // x(i)>constant
18            case "!=" => c = new DiffVal(variable, constant) // x(i)!=
    constant
19            case "<=" => c = new LeEq(variable, constant) // x(i)<=constant
20            case ">=" => c = new GrEq(variable, constant) // x(i)>=constant
21          }
22          solver.post(c)
23        } catch {
24          case _: oscar.cp.core.NoSolutionException =>
25            throw new NoSolutionException
26          case _: Inconsistency =>
27            throw new NoSolutionException
28        }
29        currentVars.map(x => x.toArray.toSet)
30      case _ => currentVars.map(x => x.toArray.toSet)
31    }
32  }
```

The first thing this function does is to observe which type of `BranchOp` it receives. Depending on that type, the function behaves differently.

If the branching operation is a `Push`, the solver of ***Oscar*** pushes the current state in its trail. For that, it simply calls the function `pushState` from the `solver` global variable. It then returns the current domains of the model.

If the branching operation is a `Pop`, the solver pops the last pushed state. As for the `Push`, it can be done by calling one function : `pop`. Since the trailing of ***OscaR*** is well done, the global variable `currentVars` is also reset to the last pushed state. Therefore, the function can immediately return the current domains.

If the branching operation is a `RestrictDomain`, the solver creates the constraint defined by this `RestrictDomain` and posts it to the solver. Remember that in ***OscaR***, the `post` function makes the solver reach fix-point. Therefore, after the post, the domains in `currentVars` have been filtered until the new fix-point and hence can be returned.
Let's see in more details how to define the constraint related to the `RestrictDomain` operation. Three pieces of information are retrieved to create the constraint. The first one is an index (`r.index`). This index indicates which variable is subjected to the constraint. The second one is a constant (`r.constant`). The third piece of information is the relation between this constant and the variable (`r.op`). Here, the six types of domain restrictions *CPChecker* is able to perform are visible in the `r.op match {...}` statement. Once those pieces of information are retrieved, the constraint related to the domain restriction is easy to create.

Choco

After the example in *Scala*, comes the one in *Java*. The *Sum* constraint of ***Choco*** is also bound($\mathcal{Z}$) consistent. Therefore, the same trusted filtering algorithm from the ***OscaR*** example is used.

```
1
2 public class SumIncrementalTest {
3
4   public static void main(String[] args) {
5     class MyFilter extends JFilterWithState {
6       private Model model;
7       private IntVar[] x;
8
```

```
9       public Set<Integer>[] setupJava(Set<Integer>[] variables) {
10        model = new Model("sum problem");
11        x = new IntVar[variables.length];
12        for (int i = 0; i < variables.length; i++) {
13          int[] b = variables[i].stream().mapToInt(Number::intValue).
14                    toArray();
15          x[i] = model.intVar("" + i, b);
16        }
17        model.sum(x, "=", 15).post();
18        Solver solver = model.getSolver();
19        try {
20          solver.propagate();
21        } catch (Exception e) {
22          throw new NoSolutionException("");
23        }
24        return transform(x);
25      }
26
27      public Set<Integer>[] branchAndFilterJava(BranchOp b) {
28        if (b instanceof Push) {
29          model.getEnvironment().worldPush();
30        } else if (b instanceof Pop) {
31          model.getEnvironment().worldPop();
32          Constraint[] cs = model.getCstrs();
33          model.unpost(cs[cs.length - 1]);
34        } else if (b instanceof RestrictDomain) {
35          RestrictDomain r = (RestrictDomain) b;
36          model.arithm(x[r.index()], r.op(), r.constant()).post();
37          try {
38            model.getSolver().propagate();
39          } catch (Exception e) {
40            throw new NoSolutionException("");
41          }
42        }
43        return transform(x);
44      }
45    }
46    FilterWithState trusted = new IncrementalFiltering(new BoundZFiltering(
    Checkers.sum(15, "=")));
47    CPChecker.check(trusted, new MyFilter(),
48          CPChecker.testArguments(), CPChecker.stats());
49  }
50
51
52  private static Set<Integer>[] transform(IntVar[] input) {...}
53 }
```

In this ***Choco*** example, everything is done in the main function. The trusted and tested filtering are created then passed in argument to the `check` function. Let's see how the tested filtering has been created.

First a new class has been created which extends `JFilterWithState`. This new class represents the tested filtering algorithm. This class possesses two variables and two functions. The two internal variables are a model used to represent the current tested model and the current domains of this model. The two functions are `setupJava` and `branchAndFilterJava`.

Let's first go over the `setupJava` function. Since this example is in *Java*, the domains are

represented as `Set<Integer>[]`. This setup function is similar to the ones representing the `filter` function of a `Filter` object in the *Java* static filtering examples (section 3.2.8). But as for the *Scala* incremental example, here the model and the domains are the global variables of the class `MyFilter` to be accessible by the `branchAndFilter` function. The model is instantiated with the variables' domains in the `setup` function to be used by the `branchAndFilter` function to perform a pseudo-search.

The `branchAndFilter` function behaves in the same way than the one of the *Scala* example. The type of the branching operation is identified, then the function adapts its behavior to this type. In ***Choco***, the trailing mechanism is accessible with the `getEnvironnement()` function of the `model`. Then, to do a push (resp. a pop), the function `worldPush()` (resp. `worldPop()`) must be called. When the branching operation is a `Push`, simply calling this function is enough for the trailing but for a `Pop`, one more change is needed. The trailing mechanism of ***Choco*** does not modify the constraints posted to the solver. However, when there is a `RestrictDomain`, a new constraint representing this restriction is added to the model. Hence this added constraint should be removed if a `Pop` happens. Hopefully, since the *CPChecker* performs dives, it is known that when a `Pop` is called, only the last restriction of domains must be removed (since when making a dive, a push is performed after each domain restriction). Therefore the constraint to be removed is the last in the array returned by the ***Choco*** function `getCstrs()`. Then to remove this constraint of the model, the `unpost` function is called over the last constraint. Once the constraint is unpost and the `worldPop` function has been called, the Pop is done. Note that if the *CPChecker* did not perform dives and the number of restrictions added between a push and a pop were unknown, it would still be possible to easily remove them by using a counter to count the number of constraints added since the last Push.

If the `branchAndFilter` function must perform a `RestrictDomain`, a constraint representing this restriction is added to the model. ***Choco*** possesses the `arithm` function. This function is able to create the constraint represented by the `RestrictDomain` in a single line. Then the solver is propagated until fix-point.

Once a branching operation had its work done in the `branchAndFilter` function, the function only needs to return the current domains as `Set<Integer>[]`. As in all other *Java* examples, the `transform` function has been used to convert the ***Choco*** current domains into the `Set<Integer>[]` type.

3.4 Custom Assertions

CPChecker allows a user to easily test a filtering algorithm over random instances. But the user may want to create unit tests with *CPChecker* using for example, *JUnit*. Unfortunately, since the errors found by *CPChecker* are reported in a file, it would not be convenient to understand the assertions error messages. To avoid this problem, *CPChecker* has created its own assertions usable in unit tests such as *JUnit* or *ScalaTest*.

Therefore, we created two classes for the assertions : `FilterAssert` and `FilterWithStateAssert`. For this, we used *AssertJ*[13], a *Java* library allowing the creation of personalized assertions. Each of the assertion classes of *CPChecker* extends the `ObjectAssert` class of `AssertJ`. This class contains useful asserts for objects. Therefore, it was extended to have the functions of *CPChecker* for the `Filter` and `FilterWithState` objects as assertions.

Those two classes have a companion object with the function `assertThat` which returns the instance of the assert class for the tested filtering algorithm given in argument.

The two classes created for custom assertions have the same functions. They are `filterAs` and `weakerThan` which represent the assertions respectively corresponding to the `check` and `stronger` functions of *CPChecker*. Hence, those functions both take in argument the trusted filtering algorithm. They also take as implicit parameter the test arguments (a `TestArgs` object) to create the random instances. Unlike the `check` and `stronger` functions of the `CPChecker` object, there was no need for a `Statistics` object since its an assertion. The only interesting point of the assertion is in its error message.

With those functions, it is possible to test the filtering algorithms in one line like this:

```
assertThat(testedFiltering).filterAs(trustedFiltering)
```

Those functions can be called successively because they return the object itself.

The `FilterAssert` and `FilterWithStateAssert` classes have exactly the same implementation with the only difference being that `FilterAssert` considers `Filter` objects while `FilterWithStateAssert` considers `FilterWithState` objects. Let's see the implementation of `FilterAssert` class and its companion object:

```
class FilterAssert(tested: Filter)
  extends ObjectAssert[Filter](tested) {

  def filterAs(trusted: Filter)(implicit testArgs: TestArgs):
   FilterAssert = {
    isNotNull
    implicit val stats: Statistics = new Statistics("")
    val result: Boolean = CPChecker.check(trusted, tested)
    var errMsg = "Tested and trusted filterings does not filter the same\n"
    //compute error message from statistics
    errMsg += stats.getErrorMsg
    val a: AbstractBooleanAssert[_] = Assertions.assertThat(result).
   overridingErrorMessage(errMsg)
    a.isTrue
    this
  }

  def weakerThan(trusted: Filter)(implicit testArgs: TestArgs):
   FilterAssert = {
    isNotNull
    implicit val stats: Statistics = new Statistics("")
    val result: Boolean = CPChecker.stronger(trusted, tested)
    var errMsg = "Tested filtering is not weaker than the trusted\n"
    errMsg += stats.getErrorMsg
    val assertion: AbstractBooleanAssert[_] = Assertions.assertThat(result)
   .overridingErrorMessage(errMsg)
    assertion.isTrue
    this
  }
}

object FilterAssert {
  def assertThat(tested: Filter): FilterAssert = new FilterAssert(tested)
}
```

The `assertThat` function of the companion object is easy to understand since, as said before, it simply returns a new `FilterAssert` object which looks at the filtering algorithm passed in argument. This function has been created to respect the conventions of *AssertJ*.

The two functions of the `FilterAssert` class only differ by calling the `check` or `stronger` function. Otherwise, they both create statistics and then obtain the error message from those. After that, comes the assertion using *AssertJ*. By calling the `overridingErrorMessage` function, the error message will be the one received from the statistics when an assertion error occurs. Of course, such an error will only occur when the comparison of the filtering algorithms has found an error.

The error message contains the first and last failed instances. They are given in the same way as those given in the files created by the `Statistics` object (see section 3.2.6 and 3.3.5).

3.5 Implemented Checkers

As explained in the *Testing Static Filtering Algorithms* and *Testing Incremental Filtering Algorithms* parts, for some constraints, checker functions have already been defined in the `Checkers` object. To recall, a checker function models a given constraint by taking an instantiation in argument and returning true if the instantiation satisfies the constraint, false otherwise. Some settings for the domains generation have also already been implemented for some constraints in the `Generators` class. This section will show the different checkers *CPChecker* possesses.

3.5.1 AllDifferent

For the *AllDifferent* constraint (for definition, see section 2.2.1), the checker function can be easily created by transforming the array into a set because a set only keeps the same elements once. Therefore, comparing the size of the original array with the size of the corresponding set will do the trick :

```
def allDifferent(instantiation: Array[Int]): Boolean =
  instantiation.toSet.size == instantiation.length
```

If the set and the array have the same size, it means that all the elements of the array are different. Consequently, the *AllDifferent* constraint is respected and the checker returns true.

3.5.2 Element

For the *Element* constraint (for definition, see section 2.2.3), this is more complicated. The first important thing to note here is that the order of the variables has its importance. The first step is to decompose the variables into three parts (X, i, v) before calling the *Element* constraint over these three parts :

```
def element(instantiation: Array[Int]): Boolean = {
  if (instantiation.length <= 2) return false
  val X: Array[Int] = instantiation.dropRight(2)
  val i: Int = instantiation(instantiation.length - 2)
  val v: Int = instantiation(instantiation.length - 1)
  element(X, i, v)
}
```

The *Element* constraint over the three parts can be simply implemented by requiring that the ith index of the X is equal to v :

```
def element(X: Array[Int], i: Int, v: Int): Boolean = {
  if (i < 0 || i >= X.length) return false
  X(i) == v
}
```

The first condition is needed to ensure that i is indeed an index of X before calling $X(i)$.

3.5.3 Sum

For the *Sum* constraint (for definition, see section 2.2.2), an operator and a constant are needed as arguments as in addition to the variables. Depending on them, a simple comparison is performed :

```
def sum(instantiation: Array[Int], constant: Int = 0, operator: String = "=
    "): Boolean = {
  if (operator.equals("=")) return instantiation.sum == constant
  else if (operator.equals("!=")) return instantiation.sum != constant
  else if (operator.equals(">")) return instantiation.sum > constant
  else if (operator.equals("<")) return instantiation.sum < constant
  else if (operator.equals(">=")) return instantiation.sum >= constant
  else if (operator.equals("<=")) return instantiation.sum <= constant
  false
}
```

3.5.4 Table

For the *Table* constraint (for definition, see section 2.2.4), the checker takes as argument the table of all possible solutions. It simply checks that the instantiation taken as argument belongs to the table :

```
def table(instantiation: Array[Int], table: Set[Array[Int]]): Boolean = {
  table.exists(x => x.sameElements(instantiation))
}
```

3.5.5 Global Cardinality Constraint

For the *Gcc* constraint (for definition, see section 2.2.6), the checker takes as arguments the instantiation, the occurrences and the values. It first checks conditions over the input variables to check they respect the correct format. Then, it creates a map and counts the number of times each value occurs in the instantiation. Finally, it checks that the count for each value is equal to the corresponding occurrence variable. Here is its detailed implementation :

```
def gcc(instantiation: Array[Int], occurences: Array[Int],
  values: Array[Int]): Boolean = {
  assert(occurences.length == values.length)
  if (instantiation.length < occurences.sum) return false
  var valuesCount: Map[Int, Int] = Map()
  values.foreach { v => valuesCount = valuesCount + (v -> 0) }
  instantiation.foreach { x =>
    if (valuesCount.contains(x)) {
      valuesCount = valuesCount.updated(x, valuesCount(x) + 1)
    }
  }
  for (i <- values.indices) {
    if (!(occurences(i) == valuesCount(values(i)))) return false
  }
  true
}
```

3.6 Software Quality

3.6.1 Correctness

In order to prove the correctness of *CPChecker*, we realized many unit tests using the `ScalaTest`[5] unit testing *Scala* framework. We modified the *build.sbt* file of our project in order to run automatically all tests at each commit on github. The build status is then displayed in the github readme :

Figure 3.5: Representation of the build status and coverage on the readme github repository

For this, we used *Travis Code Integration*[6]. We also used codecov[7] in order to compute and display the coverage reached while running the tests. Here is a representation of the coverage obtained for all parts of our code :

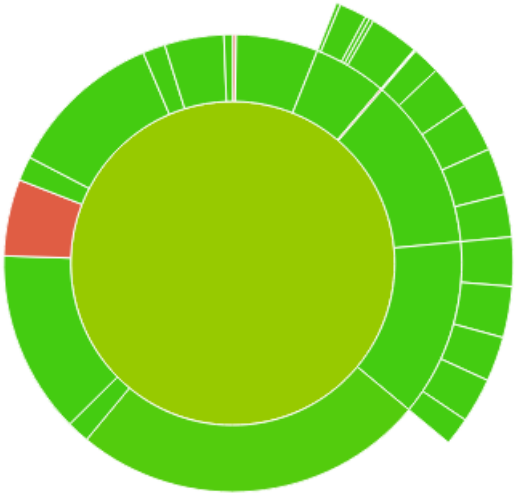


Figure 3.6: Representation of the code coverage

This figure is a graphical representation of the code coverage. The more green it is, the more the class is covered. The internal circle represents the average of the code coverage over the whole code (which has a percentage of 93% lines covered). The quarters surrounding the circle represents the decomposition into classes. The size of the quarter is proportional to the number of lines of the class. The red quarter is the `Generators` class. This class simply gives possible generators implementation for some constraints. There is nothing to test here, it only represents possible generators. So, we did not test this class. The `JFilter` and `JFilterWithState` abstract classes are not covered as well. These are simple adaptations of the `Filter` and `FilterWithState` classes containing abstract methods and so, it is not really interesting to test them. This is represented by the very thin red line on the diagram.

Since the filtering implementations of the `ArcFiltering`, `BoundZFiltering`, ... must be bug-free, we pay a particular attention to test each of them exhaustively. We tested them with several checkers, over various domains, leading to very different results.

[5] http://www.scalatest.org/
[6] https://travis-ci.org/
[7] https://codecov.io

3.6.2 Readability

The readability is a key aspect in programming. Indeed, without it, a user can experience difficulties to understand the meaning of some code fragments. Consequently, the code becomes equivalent to a black box for the user. This only lets two choices to the user : either using the tool without understanding what he is doing or not using it. These two scenarios are really bad. This is why readability is essential.

In our case, it is particularly important that the code does not look like a black box for the user. Indeed, since *CPChecker* is a tool to test and debug the user's filtering, its implementations must be trusted by the user. To obtain the user's trust, we have to be transparent. So, readability has became one of our major preoccupation.

Another reason why the readability is so important is that this tool can be extended by other developers. And without readability, most developers would not fully understand the code and *CPChecker* would not have many growing possibilities.

The code has been made readable in several ways. Firstly, we cut it into different packages to increase readability. The packages structure looks like :

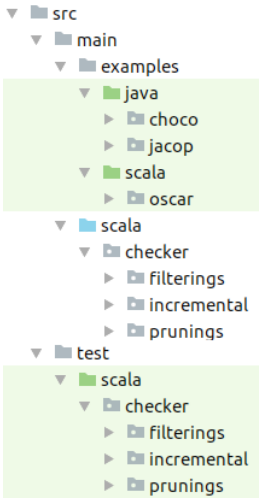


Figure 3.7: Github : structure of the repository

Some examples of *CPChecker*'s usage are in *src/main/examples/*. A user can consult them in order to know how to use this tool. There are examples for testing the ***Choco***, ***JaCoP*** and ***OscaR*** solvers. This covers examples in *Scala* and *Java* to dispose of both types of examples. The code of the tool performing the testing is located in *src/main/scala/checker*. All the unit tests are located in *test/scala/checker*. Such a package decomposition follows the sbt's conventions [8].

After that, we make the code more readable by commenting important functions that involve complex lines of code. Finally, each class has been correctly indented to improve the code comprehension.

In order to help the user use this tool, a *readme.md* file has been written. Also, to help in the understanding on how to use the tool, a user guide is available on *Github* in the *documentation/userGuide.md*. It summarizes the different things *CPChecker* can test and how to test it over detailed examples.

[8] https://www.scala-sbt.org/1.0/docs/Directories.html

Chapter 4

Performances

This chapter contains the analysis of the performances of *CPChecker*. The different filtering algorithms implemented by *CPChecker* will be run with different sizes and numbers of variables and the time taken for these algorithms to execute will be reported.
A filtering algorithm of *CPChecker* takes a checker function as argument and so, depends on this checker. Therefore, those filtering algorithms will also be run with different basic checker functions. To analyze the performances, three checkers will be used. One will always return true to observe the filtering algorithm when it does not filter any instantiation. Another will always return false to observe the filtering algorithm when it filters all instantiations until no solution remains. The last one is a checker representing the *AllDifferent* constraint. This one will be used to observe the filtering algorithms on a real constraint.

As a reminder, the different filtering algorithms of *CPChecker* are currently `ArcFiltering`, `ArcPruning`, `BoundZFiltering`, `BoundZPruning`, `BoundDFiltering`, `BoundDPruning`, `RangeFiltering` and `RangePruning`. To have an explanation of their implementation, see section 3.2.3.

4.1 Complexities of the Filtering Algorithms

Before going through the analysis of the filtering algorithms, it is useful to know their complexity. Hence, in this section, the complexities of the different filtering algorithms will be shown and explained.
Here are some variables needed to compute the complexities :

- n will represent the number of domains.
- d will represent the size of the biggest domain.
- r will represent the size of the biggest domain's interval (the domain's interval is the set containing all values in [domain.min, domain.max]).
- $c(n)$ will represent the complexity of the checker used by the filtering algorithm. Note that it depends on the number of domains.

With those variables, all the complexities can be established. First, let's display a table containing all the complexities. Then, let's explain each of them.
Here is the table containing all the complexities :

filtering algorithm	worst complexity	best complexity
ArcFiltering	$\mathcal{O}(d^n c(n))$	$\Omega(d^n c(n))$
ArcPruning	$\mathcal{O}(\sum_{i=1}^{n} d^i c(i))$	$\Omega(dc(1))$
BoundZFiltering	$\mathcal{O}(2dn^2 r^{n-1} c(n))$	$\Omega(2n * (n + c(n)))$
BoundZPruning	$\mathcal{O}(2dn^2 \sum_{i=1}^{n} r^i c(i))$	$\Omega(min(dnrc(1), 2n \sum_{i=1}^{n} c(i))$
BoundDFiltering	$\mathcal{O}(2dn^2 d^{n-1} c(n))$	$\Omega(2n * (n + c(n)))$
BoundDPruning	$\mathcal{O}(2dn^2 \sum_{i=1}^{n} d^i c(i))$	$\Omega(min(nd^2 c(1), 2n \sum_{i=1}^{n} c(i))$
RangeFiltering	$\mathcal{O}((dn)^2 r^{n-1} c(n))$	$\Omega(dn * (n + c(n)))$
RangePruning	$\mathcal{O}((dn)^2 \sum_{i=1}^{n} r^i c(i))$	$\Omega(min(drc(1), dn \sum_{i=1}^{n} c(i))$

For the `ArcFiltering`, the worst and the best complexities are the same. Indeed, it computes all the instantiations before filtering the ones that do not respect the checker function. The computation of all the instantiations takes a complexity of d^n, because each value of each domain is combined with all others to create an instantiation. This is the complexity required to effectuate the Cartesian product of the domains. When we have all instantiations, we will call the checker function over each of them. This is why the total complexity for the `ArcFiltering` algorithm is $d^n c(n)$.

Now, we will have a look at the complexities of `ArcPruning`. To recall, `ArcPruning` can only be used with anti-monotonic checkers and removes partial instantiations during the creation of them. It requires more calls to the checker but can be faster because it avoids creating useless instantiations. In the worst case, no partial instantiation is filtered and so, the complexity is even worst than the one of `ArcFiltering` since it makes more calls to the checker but still computes all instantiations. The complexity is then $\sum_{i=1}^{n} d^i c(i)$ because the creation of each partial instantiation will require a call to the checker function. For the best case complexity, all partial instantiations of size 1 will already be filtered. In this case, we will only create d instantiations of 1 value and for each of them, we will make a call to the checker function. Since these instantiations have a size 1, the complexity of calling the checker over one of them is $c(1)$. It leads us to the best case complexity of $dc(1)$.

Comparing the complexities of `ArcFiltering` and `ArcPruning`, one can remark that the worst case complexity of `ArcPruning` is bigger than the one of `ArcFiltering`. It is weird because as we explained previously, this algorithm should be faster than the `ArcFiltering` one. But in practice, this is the case. In practice, `ArcPruning` should be faster than `ArcFiltering` because even though the worst case complexity is a little bit bigger, the best case complexity is a lot smaller. So, in average, `ArcPruning` is faster than `ArcFiltering`.

For the `BoundZFiltering`, in the worst case, there is no instantiation satisfying the checker and the domains updates are such that one value is removed during each pass over all variables. In this case, for each variable's bound (minimum and maximum), we fix the value of this variable to the bound and we create all the instantiations considering the other variables' domains as intervals one by one until one respects the checker function. But in the worst case, no instantiation respects the checker. Therefore, this step is in $\mathcal{O}(r^{n-1} c(n))$. Indeed, it needs to create each combination of $n-1$ values chosen between the r possibilities offered by each interval then each of those must be passed to the checker($c(n)$). Since it tries to create a complete instantiation with the min and the max of each variable and then, goes to the next one, it means that we will do $2n$ attempts trying to create an instantiation. In the worst case, over these $2n$ attempts, only one value (min or max) has been removed from a domain. This forces us to repeat this operation dn times until having an empty domain allowing the filter function to return there is no solution.

Now, we will have a look at the best case complexity of `BoundZFiltering`. The best case happens when all instantiations are solutions. So, for the creation of an instantiation, it takes

a complexity of n because it will simply creates one instantiation of n elements. At the end, each instantiation will be passed to the checker in order to filter it if it does not respect the checker. So, we have to add to n the checker's complexity, which is $c(n)$. Since we have to find an instantiation for the minimum and maximum values of a variable, the complexity can be multiply by 2, leading to $2 * (n + c(n))$. Then, since we have to do it for each variable, we multiply it again by n, leading to the best complexity of $2n * (n + c(n))$.

For the `BoundZPruning`, the worst case complexity arises when there is no instantiation satisfying the checker and when the domains updates are such that one value is removed during each pass over all variables, as for the `BoundZFiltering`. The only difference with the `BoundZFiltering` complexity is that the generation of the instantiations involves calling the checker function each time a partial instantiation is created. Therefore, the complexity required to compute this step is $\sum_{i=1}^{n} r^i c(i)$. Finally, the worst case complexity for performing `BoundZPruning` is in $\mathcal{O}(2dn^2 \sum_{i=1}^{n} r^i c(i))$.

Now, let's elaborate the `BoundZPruning`'s best case complexity. The situation in which the best case arises is difficult to identify. It can either be when all values are filtered from the very beginning or when no value is never filtered and solutions are immediately found. Let's explore the two situations and then, the best case complexity will be the minimum between the two.

First, let's see the case where all values are filtered from the beginning. Following the `BoundZPruning` algorithm, it will make at least r calls to the checker function for each value from which an instantiation is built. So, the minimum complexity required to remove a value from one domain's variable is $rc(1)$. After having removed the minimum and maximum of the first variable, we go to the next one. So, to filter a domain entirely, we have to filter other domains as well. Hence, we repeat the step of removing a value of the domain dn times : for removing all d domain's values of the n variables (note that it should be a little bit less than n since it stops at the first empty domain but it has been rounded to n). In this case, the complexity is then $dnrc(1)$.

Secondly, let's see the case where instantiations respecting the checker are directly found. During the construction of the instantiation, we make calls to the checker function. We will call it over each partial instantiation. So, the complexity required to create an instantiation is $\sum_{i=1}^{n} c(i)$. We will check every bound of every variables. So, we will create in total $2n$ instantiations because there are 2 bounds per variable and there are n variables. In this case, the complexity is thus of $2n \sum_{i=1}^{n} c(i)$.

Grouping the two complexities together, we arrive at a best case complexity for the `BoundZPruning` of

$$min(dnrc(1), 2n \sum_{i=1}^{n} c(i)).$$

Concerning the `BoundDFiltering`, it has the same algorithm as `BoundZFiltering`, but while searching for an instantiation satisfying the constraint, it does not consider intervals but domains. Therefore, the only change that must be made to the `BoundZFiltering` complexity to obtain the `BoundDFiltering` complexity is to replace r by d each time it appears. In the same way, the `BoundDPruning` complexities can be retrieved from the `BoundZPruning` complexities.

For the `RangeFiltering`, instead of just checking the minimum and maximum of domains as in `BoundZFiltering`, we check that all values of the domains belong to a solution while considering other variables' domains as intervals. Apart from this, it is the same than for the `BoundZFiltering`. So, the factor two that appears in the `BoundZFiltering` must be changed

into d. Therefore, we reached a worst case complexity of $(dn)^2 r^{n-1} c(n)$ and a best case complexity of $dn * (n + c(n))$.

For the `RangePruning`, the worst case complexity can be modified to $\mathcal{O}((dn)^2 \sum_{i=1}^{n} r^i c(i))$ by following the same reasoning as before, changing the 2 into d. For the best case complexity now, note that the first term can be reduced. Indeed, the n is not even needed since we can filter all values of the first variable into a single pass over the first variable domain. The difference here is that we will consider all values of the first variable before going to the others, and so, in the best case, we will not have to go to the others. The best case complexity is then $\Omega(min(drc(1), dn \sum_{i=1}^{n} c(i))$.

Note that all these complexities are quite big and more elaborated algorithms could have done it in less time. So one can wonder why we do not realize more elaborated algorithms. The answer to this is that we had to make bug free algorithms, able to be trusted by the user. It is the reason why we confined us to simple versions of these algorithms, being non efficient in terms of speed/memory, but great in terms of correctness.

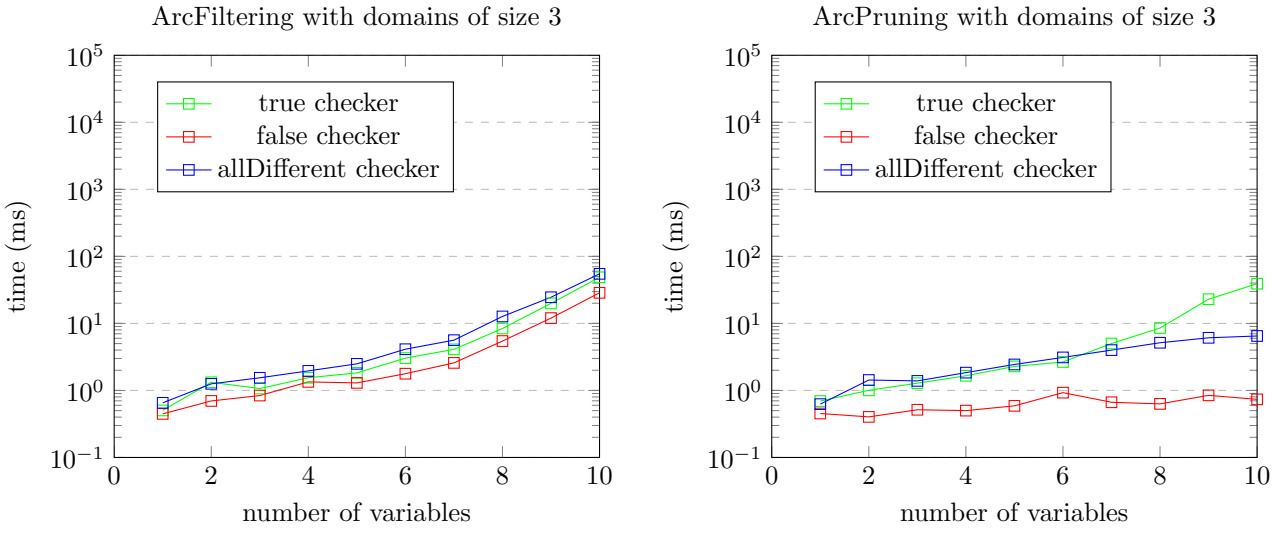
4.2 Results

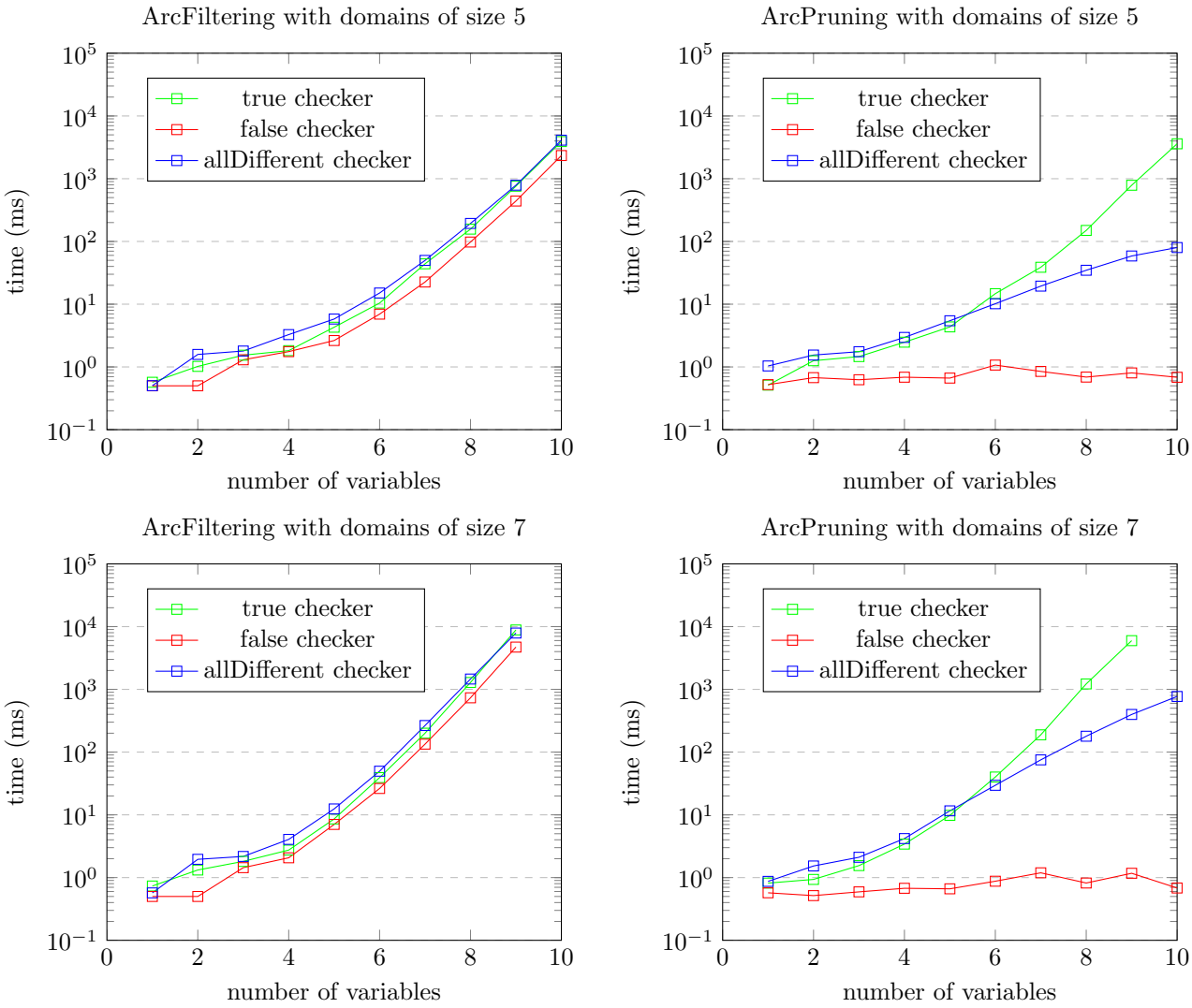
This section will show some graphs representing the time taken by CPChecker's filtering algorithms. Those algorithms will be run with domains of size 3, 5 and 7. For each of those domains' size, they will be run with 1 to 10 variables. For the analysis, the range of each variable's domain will be set between -5 and 5 included. Each measure will be computed by calling the `filter` function over 100 random instances and making the average of each function call's time. This random generation uses *Scalacheck* and the `TestArgs` class. Note that all the graphs will be obtained by running the algorithms over an Intel Core i3.

4.2.1 Arc Consistent Filtering Algorithms

In this section, the arc consistent filtering algorithms `ArcFiltering` and `ArcPruning` are analyzed.

Here are the graphs representing the time spent by the `filter` method of the `ArcFiltering` and `ArcPruning`, depending on the number of variables. There are several graphs, for different domains' sizes. On each graph, there are 3 curves representing different checkers. On the left are the graphs for the `ArcFiltering` class and on the right, the graphs for the `ArcPruning` class.





In those graphs, the impact of the size of the domains can clearly be seen. With domains of size 3, `ArcFiltering` always takes a time inferior to 100 ms but for domains of size 5 or 7, the maximum time it takes to run the `filter` function is close to 10000 ms. That is 100 times slower just by nearly doubling the size of the domains.
In truth, the domains of size 7 should have taken more time for 10 variables. The problem is that the `ArcFiltering` takes a lot of memory. This is due to the fact that the filtering algorithm of this class computes all the instantiations. Therefore, for 10 variables with a domains' size of 7, there are 7^{10} instantiations in memory before being able to reduce that number by filtering. This huge number caused an *out of memory* exception. This problem could be avoided by making the implementation of the filtering algorithm of `ArcFiltering` more efficient but it will not be done. One objective of this filtering algorithm is to be trustable. A user should be able to read it and think it works correctly. If the implementation is modified, it will become a lot more complex and therefore, stray away from its objective. If the user wants a fast and memory efficient algorithm, he can still use his own algorithm.

The graphs for the `ArcFiltering` class have similar curves for different checkers. Nevertheless, the *allDifferent* checker has a little worse complexity than the two others therefore, he is always slower. Also, the *false* checker is always the fastest one. This is because the checker filters all the solutions and therefore, the domains are not recreated from the solutions. This is that little time gained which makes it the fastest of the three checkers for `ArcFiltering`.

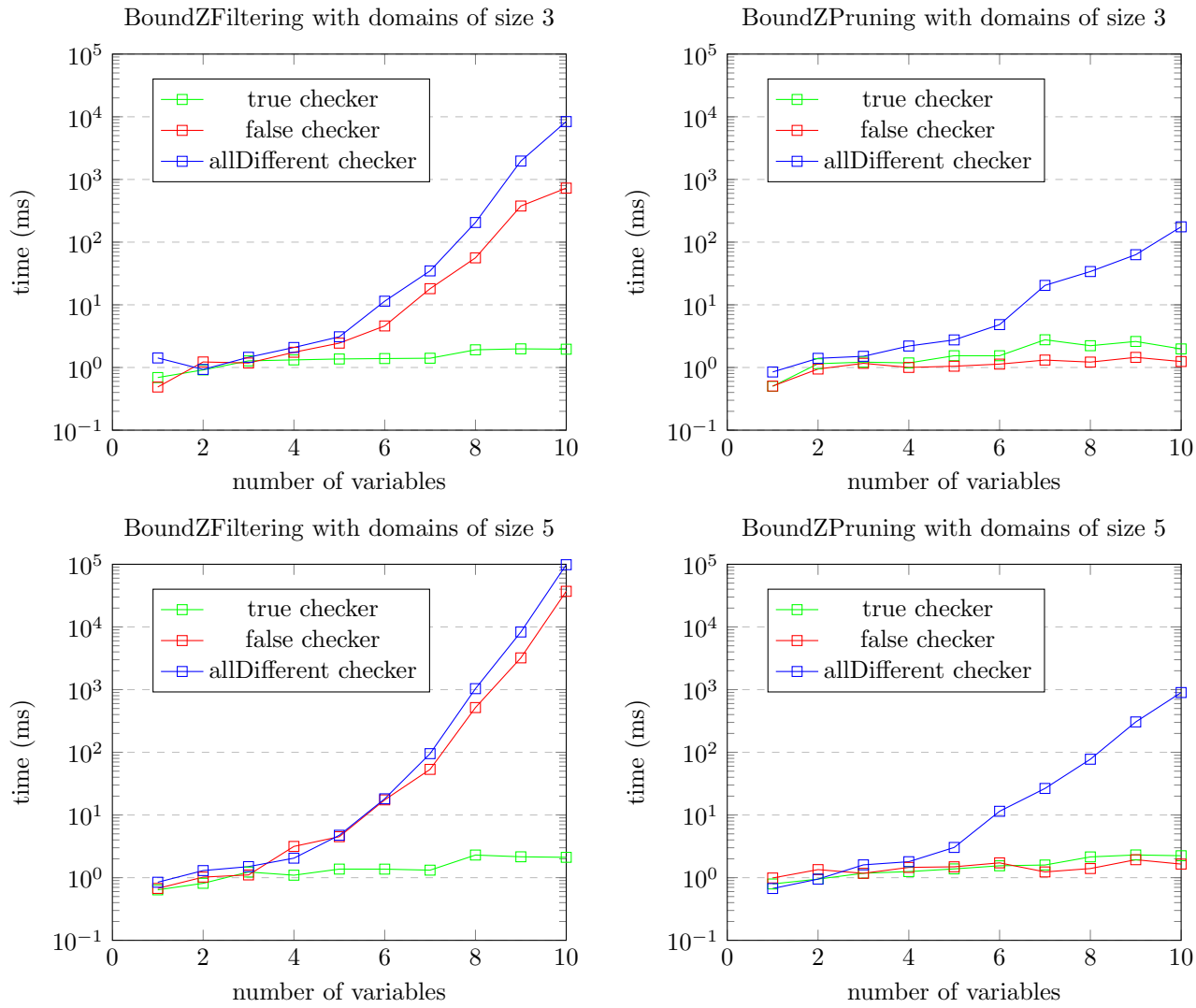
`ArcPruning` has a bigger worst case complexity than `ArcFiltering`. Nonetheless, `ArcPruning` is a lot better. Those graphs clearly show its superiority. Only the *true* checker makes `ArcPruning` behave like `ArcFiltering` because no solutions are filtered with this checker.
All the other checkers used make `ArcPruning` faster than `ArcFiltering`. In fact, it even allowed the *false* and *allDifferent* checkers to run with 10 variables with domains' sizes of 7 without causing an *out of memory* exception.

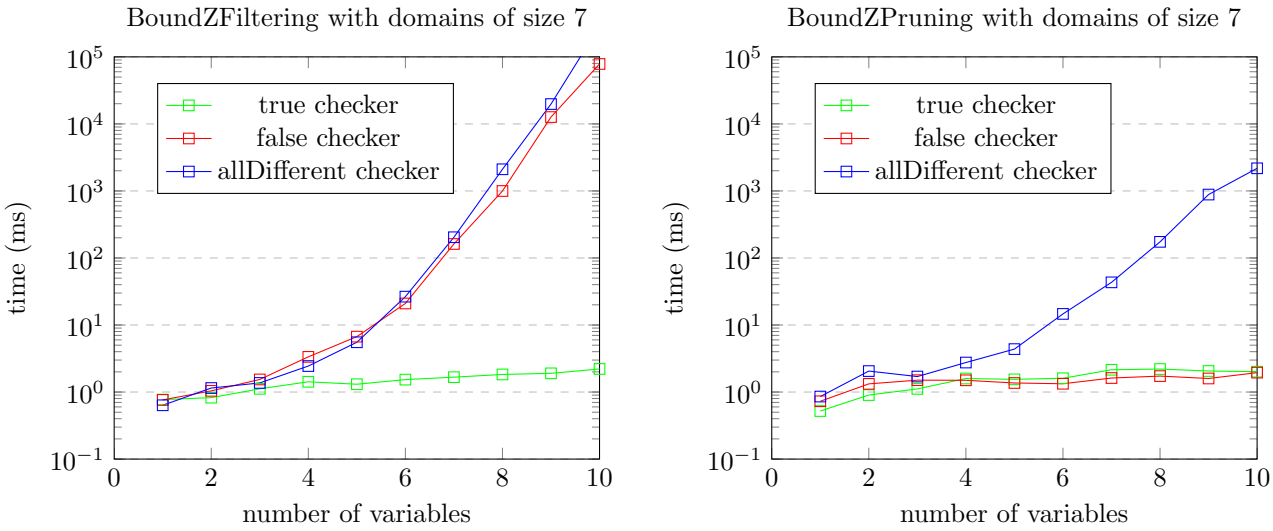
A good observation is to notice that the more often the checker returns false, the faster `ArcPruning` will be. In fact, if the checker returns nearly always true, if it is a slightly complex checker, it will become more efficient to use `ArcFiltering`. But the *false* checker shows that `ArcPruning` has a low complexity with it. It stays close to 1ms to filter the domains.
The *allDifferent* checker shows a good in-between the *true* and *false* checker. With `ArcPruning` it is approximately 10 times faster than with `ArcFiltering`.

This concludes the comparison of the arc consistent filtering algorithms.

4.2.2 Bound($\mathcal{Z}$) Consistent Filtering Algorithms

Here are the graphs obtained from the Bounds($\mathcal{Z}$) algorithms. On the left are the graphs for the `BoundZFiltering` class and on the right, are the graphs for the `BoundZPruning` class.





The execution time of the `BoundZFiltering`'s filtering algorithm depends heavily on the checker. In fact, it is faster for two possible conditions: there is a small number of values that can be filtered (many of them belong to solutions) or all the values to filter are rapidly filtered.

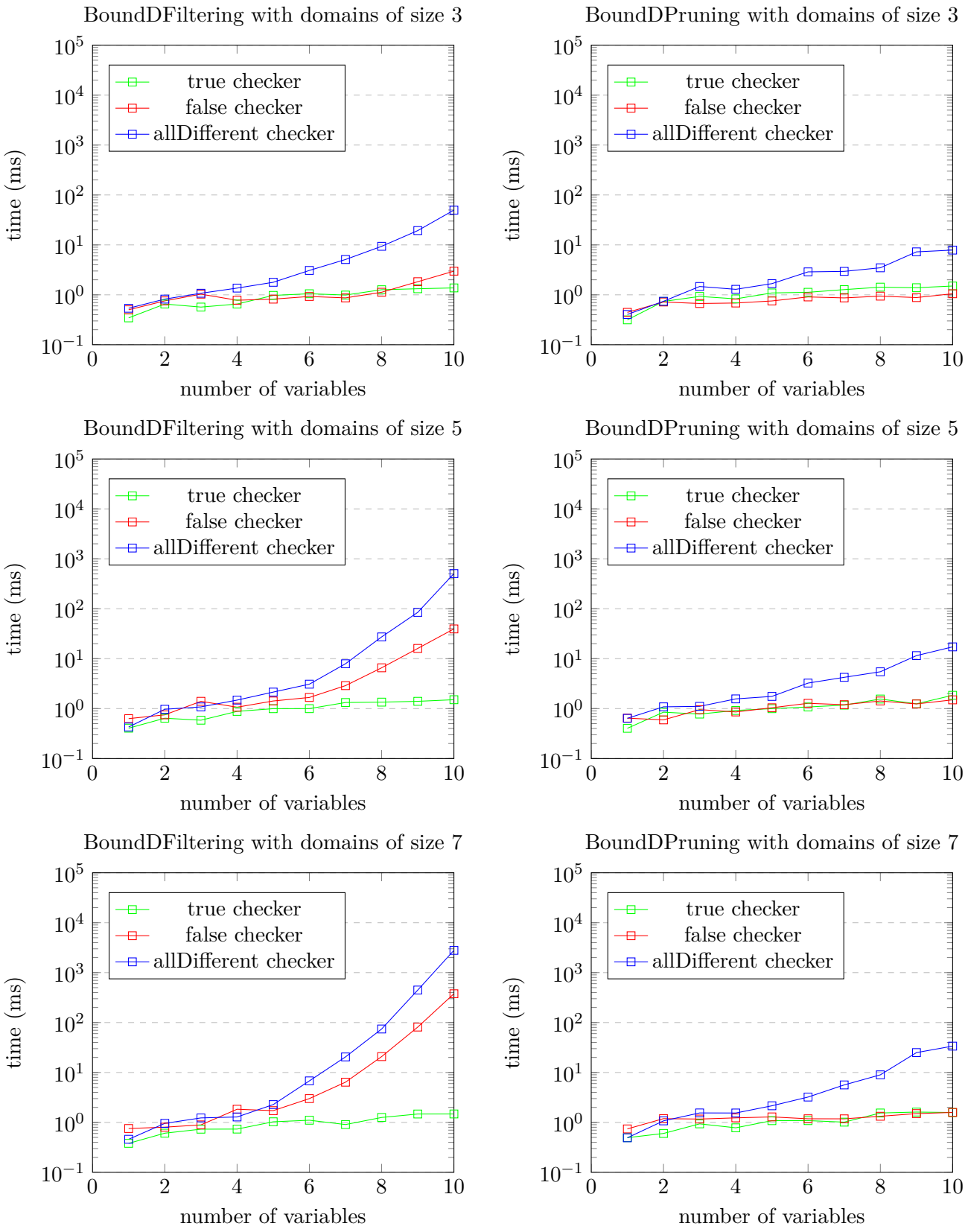
The first condition is a lot more important than the second. For example, the *true* checker has no value to remove. This is why it is fast compared to the other two. It only slightly increases with the number of variables. In comparison, the two others increase exponentially.

The second condition only explain why the *false* checker is below the *allDifferent* checker. The *false* checker only needs to look once at a value before taking it off. In contrast, the *allDifferent* checker will often need to pass again on the same values before filtering them. Nevertheless, the *allDifferent* does not always filter all the values and therefore, it is not that far away from the *false* checker.

The `BoundZPruning` is more powerful than `BoundZFiltering`. When it uses the *allDifferent* checker, it is always faster. This is because it more rapidly founds if a value must be filtered or not.
But its true worth is seen with the *false* checker. The *false* checker offers the best complexity for the `BoundZPruning` because it rapidly finds that a value must be filtered. This makes `BoundZPruning` with the *false* checker as fast as with the *true* checker while `BoundZFiltering` with the *false* checker was only slightly better than with the *allDifferent* checker.
This concludes the comparison of the bound($\mathcal{Z}$) consistent filtering algorithms.

Bound($\mathcal{D}$) Consistent Filtering Algorithms

We will now consider the Bounds($\mathcal{D}$) algorithms. Once again, the left graphs are the ones for the `BoundDFiltering` class and on the right, are the graphs for the `BoundDPruning` class.



The bound($\mathcal{D}$) consistent filtering algorithms behaves exactly like the bound($\mathcal{Z}$) ones. The difference is that it only looks at domains instead of intervals to compute instantiations.

For the `BoundDFiltering`, the *true* checker is the best and the *false* checker is slightly faster than the *allDifferent* checker. For the `BoundDPruning`, the *false* and *true* checkers have a similar

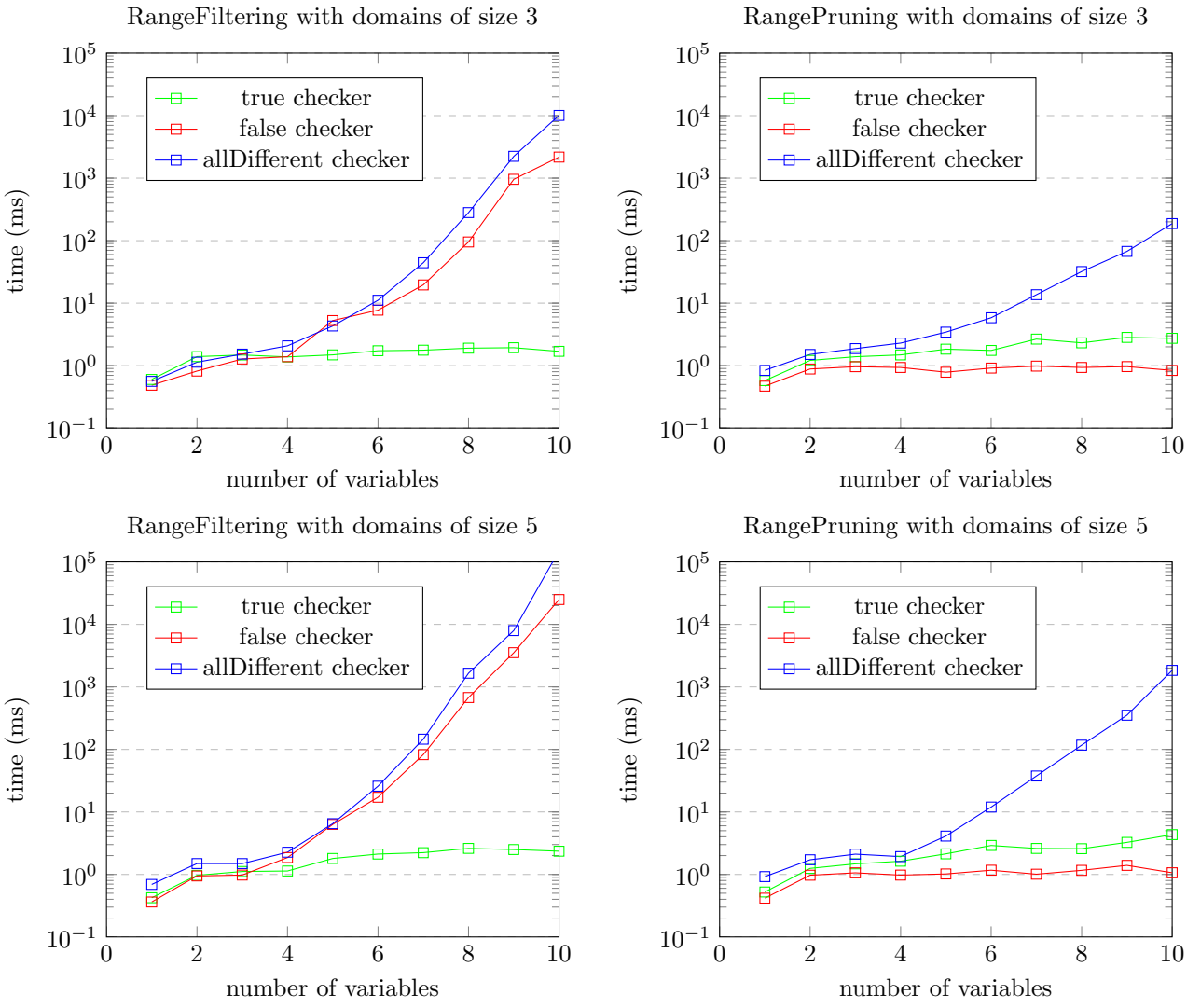
complexity. From the section on the complexities, it had been observed that the best complexity for the `BoundDPruning` was a minimum between two complexities. The *true* and *false* checkers each behaves like one of these.

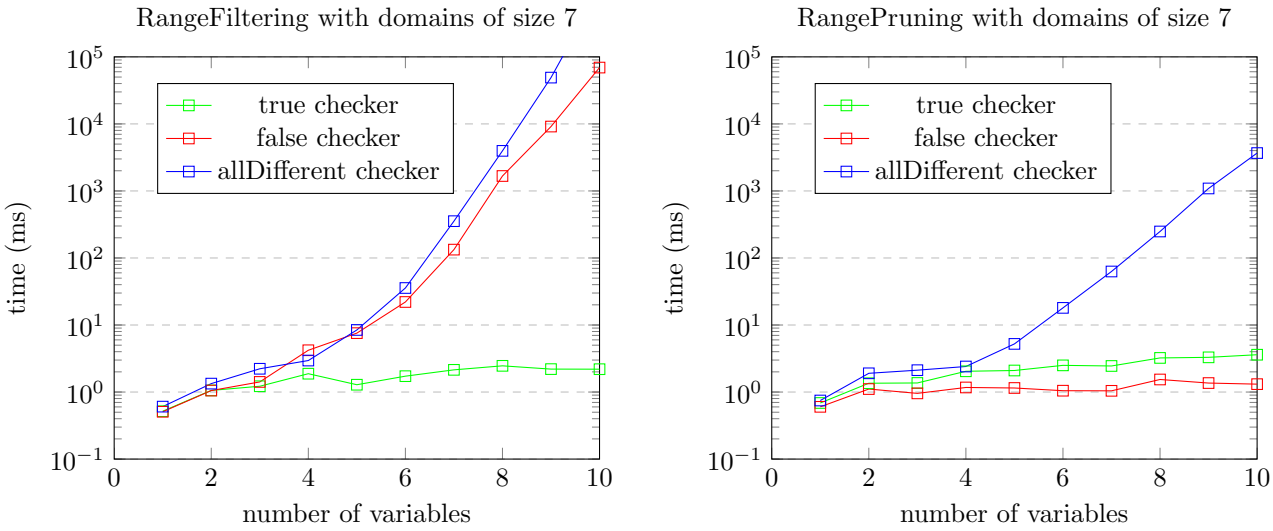
The pruning algorithm still performs better than the filtering one as observed for the previous consistencies.

4.2.3 Range Consistent Filtering Algorithms

In this section, the range consistent filtering algorithms `RangeFiltering` and `RangePruning` are analyzed.

Here are the graphs : on the left, the ones for the `RangeFiltering` class and on the right, the ones for the `RangePruning` class.





The range consistent filtering algorithms work in the same way as the bound consistent filtering algorithms.

For the `RangeFiltering`, the *true* checker gives the best complexity and therefore, is quicker than the two others.
The *false* checker is still a little faster than the *allDifferent* checker because it never goes twice over the same value.

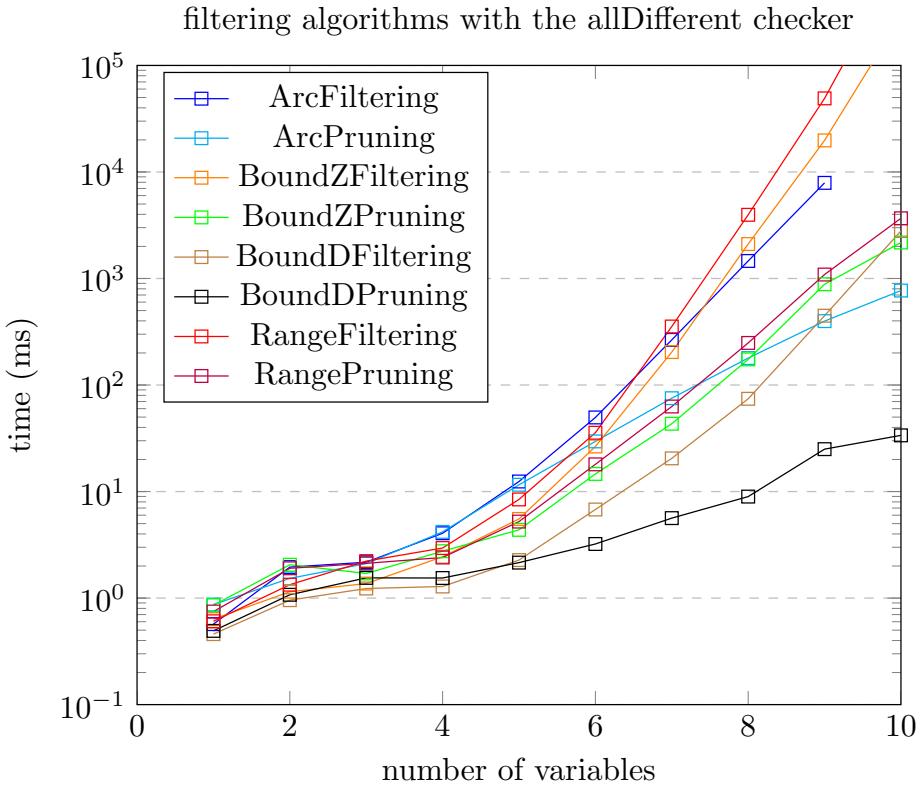
For the `RangePruning`, it is faster overall than `RangeFiltering` but the biggest change is for the *false* checker. With it, it becomes efficient and mostly, constant for a given size of the domains. In fact, the `RangePruning` with the *false* checker will only go once on each value of the first domain before finding that there is no solution. It will also be faster than the *true* checker to find if a given value must be removed. This is because the *true* checker must build a complete solution while the *false* checker only observes all the values of one interval before knowing that the value must be removed or not. All those points explain why the `RangePruning` with the *false* checker is faster than the `RangePruning` with the *true* checker.

This concludes the comparison of the range consistent filtering algorithms.

4.2.4 Filterings Comparison

In this section, all the filtering algorithms will be compared for a given size of the domains and a given checker in one graph. This will allow a concrete comparison between the performances of the different consistencies. The size of the domains will be fixed at 7.

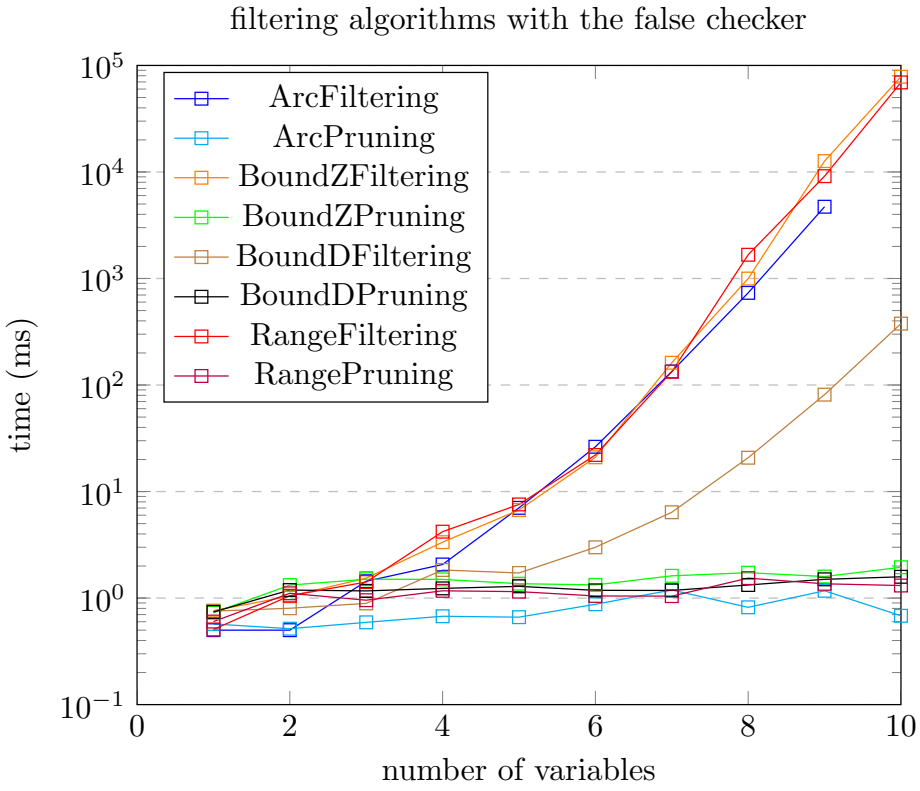
The first checker used for the comparison is the *allDifferent* checker. Here is the graph obtained:



On this graph, we can see that each `Pruning` curve is situated under the corresponding `Filtering` curve, especially for big number of variables. This is what was expected to see as the `Pruning`'s computes less instantiations than the `Filtering`'s, particularly if the checker returns often false. The *allDifferent* checker will return more often false when the number of variables is high, which explains the big difference with the `Filtering`'s in this case.

The bound($\mathcal{D}$) consistent filtering algorithms are better than the others. In fact, the way the bound consistent and range consistent algorithms are implemented is in practice often more efficient than the way the arc consistent filtering algorithms have been implemented. Nevertheless, the bound($\mathcal{Z}$) and range ones are slower than the arc consistent ones. This is because they consider intervals. The number of instantiation when considering intervals instead of domains can easily explode. For example, you could have the domains $\{1, 10\}$ and $\{2, 9\}$ which have 4 possible instantiations while their intervals have 80 possible instantiations. The difference is clear and since bound($\mathcal{D}$) considers domains, it is the fastest. Be it by comparing the `Pruning` or `Filtering` algorithms. This is for a general checker. We will display later the same kind of graphs but for very specific checkers and we will see that it is not always the same ranking. It depends a lot of the checker.

The second checker used is the *false* checker.

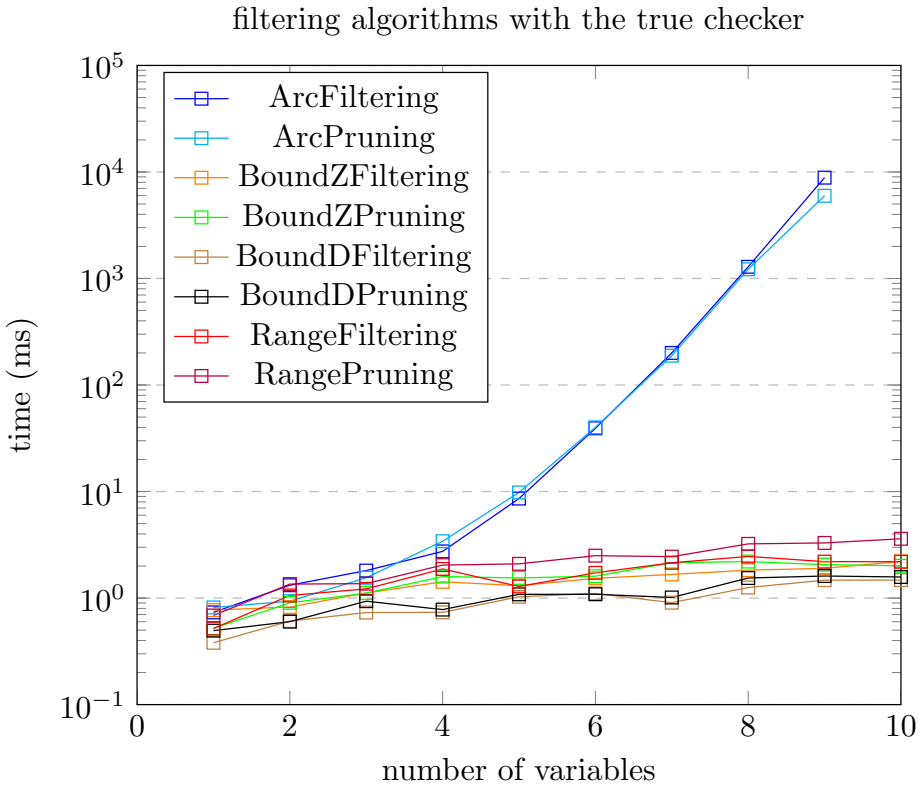


On this graph, two different behaviors can be discerned.

All these algorithms can be split in two categories: the `Filtering` ones (whose names end with '*Filtering*') and the `Pruning` ones. The `Filtering` ones must compute a lot of instantiations to know there is no solution. Therefore, those algorithms can take quite a time to execute for a big number of variables. It can be seen on the graph that those filtering algorithms grow exponentially in function of the number of variables. This is normal because increasing the number of variables augment exponentially the number of possible instantiations. This is the first behavior. From the four algorithms respecting it, the `BoundDFiltering` is the fastest. It is faster than the `RangeFiltering` and `BoundZFiltering` because compared to those, it will not consider other variables as intervals, which will restrict the number of instantiations to consider. Comparing it to the `ArcFiltering` implementation, it will consider many instantiations but not all of them and therefore, it will be faster.

The second behavior concerns the `Pruning` algorithms. Since with those algorithms, incomplete instantiations are already filtered, they swiftly find that there is no solution. Therefore, those algorithms are quite fast. They all form nearly constant curves that nearly overlap.

Now, we reported times for all these algorithms while considering a checker returning always true:



On this graph, all the algorithms have quite the same curves except the arc consistent ones. It is because of the fact that the `ArcPruning` and `ArcFiltering` algorithms will both, in this case, compute all instantiations. Indeed, since this is a checker returning always true, the gain obtained by filtering partial instantiations that do not respect the checker is null. So, `ArcPruning` will take a big time to execute as does the `ArcFiltering`.

All other algorithms are a lot faster. This is due to the fact that to attain bound or range consistencies, we do not compute all instantiations. To test that a value must be filtered or not, we will try to find an instantiation with this value respecting the checker. But since the checker returns always true, this is straightforward and takes a small time to be done. It is the reason why all these algorithms are very fast, compared to the arc consistent ones.

Chapter 5

Further Work

In this chapter, an analysis of the future of *CPChecker* has been realized. Which directions could it take? What extensions could be brought? What improvements of the existing features could be done? How could this tool become more user friendly? A lot of questions about the future of the tool need to be asked and pondered about. It allows the tool to stay up-to-date and increases its quality overall. For all those reasons, many possible extensions have been thought for this already great tool.

5.1 Scheduling

In Constraint Programming, a large number of the problems solved are problems of *Scheduling*. This type of problems is so often used that a lot of constraints specific for them have emerged such as the *Unary Resource*, the *Cumulative* or the *Energetic Reasoning*. But which particularities have those constraints? In *Scheduling*, the variables are grouped together to represent activities. One activity is represented by three variables linked by a constraint: $s + d = e$, where s represents the start time of the activity, d its duration and e its end time. The particularities of *Scheduling* constraints is that they always suppose that the variables they receive represent activities. Therefore, to test a *Scheduling* constraint, it is needed to add the constraints creating those activities. For now, *CPChecker* only allows filtering algorithms to take an array of sets of integers to represent the variables. To transform the array of domains to an array of activities or to split them between *start*, *duration* and *end* variables would ask more than a little work for the user. Therefore, it could be interesting to add a specific interface for *Scheduling* constraints.

This interface should be able to perform the same things as of now but it would for example, not take one array of domains but three: one for each type of variables (*start*, *duration* and *end*). This would need a lot of work to integrate but it would really ease a lot the workload of a user testing *Scheduling* constraints. Since it is one of the objective of *CPChecker*, it would be a good improvement.

5.2 Stronger to Faster

For now, *CPChecker* only contains two functions for the testing: `check` and `stronger`. These two functions can help to test if a filtering algorithm filters correctly. But what about their performances? In practice, it is not the strongest that is preferred but the one which permits the solver to be the fastest. Therefore, a good improvement of the software could be to add a `faster` function in addition to the `check` and `stronger` functions. This `faster` function would check if the tested algorithm is faster than the trusted algorithm (while still returning the correct

filtered domains).

Adding this function would add an entirely new perspective for *CPChecker*. Currently, it does not care at all about the complexity of the filtering algorithms since it is outside of its scope. It can only be used to test correctness. The `faster` function would make *CPChecker* able to test the performances, the quality of a filtering algorithm. It would allow the user to easily know which filtering algorithm he should be using by comparing them.

For this `faster` function, a lot of work would be required. Instead of stopping when a test failed, this function would maybe prefer executing all the tests even if the tested filtering algorithm is sometimes slower than the trusted filtering algorithm. An other modification could be the generation of random domains. To test the performances, the best practice is to test with arguments of growing sizes. But, in *CPChecker*, the random domains are generated given a fixed number of variables with a fixed domain's size. the `TestArgs` class should then be more flexible to allow growing sizes of the random domains.

This improvement is interesting for the future of this tool. Nevertheless, it is a completely different approach compared to the current approach. All the other possible improvements still repose on the concept of correctness testing. Only this one goes beyond this concept to allow a new feature for this tool on the concept of quality testing.

5.3 Improvements of Implemented Features

In this section, some possible improvements of the currently existing features will be discussed.

The most notable changes/improvements that could be done are on the `RestrictDomain` class representing the branching operation filtering some values of a single domain. The most simple improvement would be to add more types of restriction such as, for example, the modulo to only filter odd numbers. But this improvement could not be a good change. In fact, the more restrictions of domain the class can perform, the more the user/solver must be able to do in order to test incrementally a filtering algorithm. The solver could then even have missing implementations of some restrictions which would ask the user to implement them to use *CPChecker*. Therefore, simply adding new types of restriction of domains could be a bad change. But another change could solve this issue. It could be made possible to set which types of restriction of domains can be represented by the `RestrictDomain` class during an incremental test. This change would allow us to make the `RestrictDomain` class being able to represent as many restrictions of domains as we want. But this additional flexibility could at the end, only add more to remember for the user while it does not offers a lot. In fact, the restriction of domain performed never had a big impact. The most useful thing about incremental testing was the updating of the internal state of the filtering algorithms. Therefore, this improvement could be interesting but it is not a needed improvement.

Other possible improvements of the current *CPChecker* are simply improvements that can be done little by little. For example, for now, *CPChecker* possesses filtering algorithms for the arc, bound and range consistencies. But there exist more consistencies than that. To cite two, there also are the path consistency and node consistency. There is also the *Forward Checking* when the filtering algorithm filters only after the domains have been filtered by other constraints until a certain point. Those types of filterings could be added little by little. To continue on those little improvements, more checker functions could be implemented in the `Checkers` object.

In conclusion, as for all existing softwares, a lot of possible improvements can always be done.

But each of them must be analyzed correctly because a change added to a software without thought could worsen its quality. Improving a software is what makes it lives for a long time and should constantly be sought after.

Chapter 6

Conclusion

The main challenges that we faced were to be generic and bug-free. We succeeded in them. Indeed, *CPChecker* is generic since it can be used to test a filtering algorithm for any constraint over integer domains. Furthermore, it is bug-free since the filtering implementations involve simple algorithms created to be understandable.

But being bug-free has a disadvantage : it limited us in terms of performance. Indeed, a bug-free algorithm requires to be simple to be readable and so trustable. But simplicity is often not compatible with efficiency. So the filtering algorithms implemented by *CPChecker* require to be called over quite small instances, as seen in the *Performances* chapter. Nevertheless, those small instances are enough to test most of the constraints thoroughly.

The usefulness of *CPChecker* has already been demonstrated over three concrete solvers: ***Choco***, ***OscaR*** and ***JaCoP***. Even complex constraints such as the *Cicrcuit* or the *Global Cardinality* Constraints can be tested with less than 50 lines of code. Therefore, the workload of the developers can be efficiently lightened by using *CPChecker*. This was one of the objectives of the tool which is now attained.

In the end, *CPChecker* has been written in *Scala* and has been made fully compatible with *Java* code so that any *Scala* or *Java* developer can use his favorite language to test his solver. Moreover, all solvers written in languages based on the JVM should be able to be tested using this tool. The user would simply need to implement a conversion of the *Scala* object to the desired language. Also, many unit tests have been written to measure the code quality and a *ReadMe* and documentations have been made to help the developer. The code of the tool is publicly available in our *Github* repository[1].

Nevertheless, a software must always thrive for new features or improvements. Those were analyzed in the *Further Work* chapter. Some of the possible changes would ask for heavy workloads but the core of *CPChecker* is already done and is fully functional. All those changes are extensions or improvements which do not imply changes to the core implementation of the current *CPChecker*.

[1] `https://github.com/vrombouts/Generic-checker-for-CP-Solver-s-constraints`

Bibliography

[1] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[2] Krzysztof Kuchcinski, Radoslaw Szymanek and contributors. Jacop solver. Available from `https://osolpro.atlassian.net/wiki/spaces/JACOP/`.

[3] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[4] Micha Meier. Debugging constraint programs. In *International Conference on Principles and Practice of Constraint Programming*, pages 204–221. Springer, 1995.

[5] Christian Schulte. Oz explorer: A visual constraint programming tool. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 477–478. Springer, 1996.

[6] Frédéric Benhamou and Frédéric Goualard. Debugging constraint programs by store inspection. *Deliverable available at http://discipl. inria. fr/deliverables2. html, DiSCiPl ESPRIT project*, 22532, 1999.

[7] Chiu Wo Choi, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J Stuckey. Finite domain bounds consistency revisited. In *Australasian Joint Conference on Artificial Intelligence*, pages 49–58. Springer, 2006.

[8] Yong Lin and Martin Henz. Recollection: an alternative restoration technique for constraint programming systems. *arXiv preprint arXiv:1601.07300*, 2016.

[9] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In *Foundations of Artificial Intelligence*, volume 2, pages 169–208. Elsevier, 2006.

[10] Broderick Crawford, Carlos Castro, Eric Monfroy, and Nibaldo Rodriguez. Solving constraint satisfaction puzzles with constraint programming. 2009.

[11] ScalaCheck. `http://scalacheck.org/`.

[12] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 204–212, New York, NY, USA, 2008. ACM.

[13] AssertJ Library. `http://joel-costigliola.github.io/assertj/index.html`.

[14] ScalaCheck user guide. `https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md`.

[15] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. How efficient is a global constraint in practice? *Constraints*, 23(1):87–122, January 2018.

[16] Interop between Java and Scala. http://www.codecommit.com/blog/java/interop-between-java-and-scala.

[17] Converting Java collections to Scala. http://blog.madhukaraphatak.com/converting-java-collections-to-scala/.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**