|  |
| --- |
| [Versione 1.0] |



WISE CRUNCH TOOLS USER MANUAL

Version 1.0

**Date:** 20/02/2014

**Version:** 01

Contenuti

[Combinatorial Test 3](#_Toc380678888)

[1-wise testing. 4](#_Toc380678889)

[2-wise testing o pairwise testing 5](#_Toc380678890)

[n-wise testing with n>2 8](#_Toc380678891)

[Wise crunch tools 9](#_Toc380678892)

[Installation procedure 10](#_Toc380678893)

[Configuration procedure 12](#_Toc380678894)

[Overview of WiseCrunchTools 13](#_Toc380678895)

[Lexical and typographical conventions 16](#_Toc380678896)

[First level tools – batch DOS script 17](#_Toc380678897)

[Tool runW 17](#_Toc380678898)

[Tool runCC and runsCC 17](#_Toc380678899)

[Tools runT and runsT 18](#_Toc380678900)

[Tools runTS and runsTS 19](#_Toc380678901)

[Tools runTSF and runsTSF 20](#_Toc380678902)

[Tool runR 21](#_Toc380678903)

[Tool runC 21](#_Toc380678904)

[Second level Tools – executable 24](#_Toc380678905)

[Executable calcolacopertura.exe and calcolaCoperturaSlow.exe 24](#_Toc380678906)

[Executable Combinazioni\_n\_k 24](#_Toc380678907)

[Executable generaTestSet.exe and generaTestSetSlow.exe 25](#_Toc380678908)

[Executable ProdCart.exe 26](#_Toc380678909)

[Executable reduceNple.exe 27](#_Toc380678910)

[Utility executable 28](#_Toc380678911)

[Document’s revision 30](#_Toc380678912)

# Combinatorial Test

When a software application accept several inputs, each of which can assume different values, it is impossible - in general - to test all combinations of values of input variables, simply because they are too many. Let's make an example, consider a software feature that accepts as input three possible values A, B and C. These values can be chosen arbitrarily from the following table:

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| A1 | B1 | C1 |
| A2 | B2 | C2 |
| A3 | B3 |  |
| A4 |  |  |
| # Valori | | |
| 4 | 3 | 2 |

**Tabella 1 – Variables and values**

The total number of possible combinations of variables (A, B, C) is equal to ; in practice, in order to ensure trying at least once all possible combinations of the values of the variables (A, B, C) 24 test cases must be carried out. Such combinations are the following:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **1-4** | **5-8** | **9-12** | **13-16** | **17-20** | **21-24** |
| A1;B1;C1 | A1;B3;C1 | A2;B2;C1 | A3;B1;C1 | A3;B3;C1 | A4;B2;C1 |
| A1;B1;C2 | A1;B3;C2 | A2;B2;C2 | A3;B1;C2 | A3;B3;C2 | A4;B2;C2 |
| A1;B2;C1 | A2;B1;C1 | A2;B3;C1 | A3;B2;C1 | A4;B1;C1 | A4;B3;C1 |
| A1;B2;C2 | A2;B1;C2 | A2;B3;C2 | A3;B2;C2 | A4;B1;C2 | A4;B3;C2 |

**Tabella 2 – A, B and C values’ variables combinations**

Now, in this particular case, such number of tests can still be affordable. However, if we consider the general case of N variables *X1, X2, …Xk* , the first accepting *n1* possible values, the second *n2* possible values, the k-th that assumes *nk* possible values, the total number of combinations is equal to: . Such a value, even for low values of *n1*, *n2* ,…, *nk* is an high number. For example, if k = 5 and (*n1*=3; *n2*=4; *n3=2;* *n4=2;* *n*5=3) we get a combinations’ number equal to  that is quite a large number of tests to perform if you want to ensure complete coverage of all combinations.

In real software applications, the number of values that can assume ni variables is high and it's easy to reach the hundreds of thousands (or millions) of combinations, which makes it impossible to perform comprehensive tests on all combinations.

How can we carry out an effective test when the number of variables and values is so high to make it impossible to exhaustively test all combinations? What reduction techniques apply?

# 1-wise testing.

When combinations’ number is high, it is possible at least verify that - at least once - each individual value of the variables is given as input to the program to be tested. In other words, if the variable A can take the values A1, A2, A3, it must be executed at least a first test in which the variable A = A1, a second test in which A = A2, and a third test in which the variable A = A3; the same goes for the other variables. This type of test provides a so-called wise-1 cover, and we will see shortly the meaning. In practice, we have the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **# TEST** | **A** | **B** | **C** |
| 1 | A1 | \* | \* |
| 2 | A2 | \* | \* |
| 3 | A3 | \* | \* |
| 4 | A4 | \* | \* |
| 5 | \* | B1 | \* |
| 6 | \* | B2 | \* |
| 7 | \* | B3 | \* |
| 8 | \* | \* | C1 |
| 9 | \* | \* | C2 |

**Tabella 3 – Max wise-1 Test Set**

A first reduction that is possible to do is to set a value for the first variable and assign a random (but permitted) value to the other variables (stated with \* in **Tabella 3**) and proceeding in this way for all the variables and values. In this way we reduce the test cases from 24 to just 9. It’s still possible to further reduce the number of test cases considering that instead of \* you can put a value of the variable which can then be excluded from the subsequent test cases.

Put into practice, for test case # 1 in place of B = \* put B = B1, instead of C = \* put C = C1 and remove test case # 5 and test case # 8, which –now - are both covered by the test case # 1;

Test case # 2: in place of B = \* put B = B2 and n place of C = \* put C = C2 and erase test cases # 6 and # 9 both of which are now covered by the test case # 2 .

Test case # 3: instead of B = \* put B = B3 and in place of C = \* insert any value C1 or C2, considering that the values of the variable C equal to C1 and C2 are already been covered by test cases # 1 and # 2; we can let C=\* and postpone the choice of whether to enter C1 or C2. Now, remove test case # 7, since B = B3 is now covered by the test case # 3.

Having understood the mechanism, there remains only the test case # 4, which covers A = A4; we can let B=\* and C=\* postponing the choice of what to actually select when we will really perform the test.

The symbol \* represents the "don’t care"; any value we put in it, the coverage of the test set does not change and all variables values will be used at least once. Those with "\*" value ​​should be covered more than once.

The final minimized Test Set for wise-1 coverage is the following:

|  |  |  |  |
| --- | --- | --- | --- |
| **# TEST** | **A** | **B** | **C** |
| 1 | A1 | B1 | C1 |
| 2 | A2 | B2 | C2 |
| 3 | A3 | B3 | \* |
| 4 | A4 | \* | \* |

**Tabella 4 – Minimized wise-1 Test Set**

is drawn from moving up the columns of the variable B and C to fill the values ​​with \* with specific values; the \* values “stagnate” in the lines that cannot be covered from specific values ​​(row 3 variable C and row 4 variable B), because the number of values of the variables are different (for example, variable B has just 3 values, while variable A has 4 values; the missing B value – with respect to A – is replaced by \*).

Therefore saying that a test set, such as that reported in **Tabella 4** , provides wise-1coverage is equivalent to say that each individual value of each variable is covered at least once.

The general wise-1 rule we can deduce is the following:

“N variables *X1, X2, …Xk*, the first assuming *n1* possible values, the second *n2* possible values, the k-th *nk* possible values, the maximum number of tests that provide coverage wise-1 is equal to , while the minimum number of tests is equal to the maximum value among { *n1* , *n2* ,…, *nk* }.”

In real cases, what is of interest is always the Test Set with the minimum number of test cases ensuring the chosen coverage (and this for obvious reasons).

# 2-wise testing or pairwise testing

If the wise-1 test guarantees coverage of every single value for each variable, it is easy to see that a wise-2 test set ensures that all pairs of values of the variables are covered at least once. In the case of the variables listed in **Tabella 1**, all **pairs of variables** are as follows: {(A, B), (A, C), (B, C)}. In fact, the combinatorial calculation shows that the number of combinations of N values taken K to K (with N ≥ K) is equal to:



In our example we have three variables (N=3) taken two to two (K=2), an applying the combinatorial formula above we get; the three pairs are {(A, B), (A, C), (B, C)}.

Wanting to compute all possible **pairs of** **variable values,** we need to consider the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **COPPIA** | **# VALORI VARIABILI** | | | **TOTALE PARZIALE** |
| **A** | **B** | **C** |
| (A,B) | 4 | 3 |  |  |
| (A,C) | 4 |  | 2 |  |
| (B,C) |  | 3 | 2 |  |
| **TOTALE COMPLESSIVO** | | | |  |

**Tabella 5 –** Counting of the pairs of values of the variables A, B and C

Hence, the total of all the pairs of values of the variables A, B and C whose values are reported in **Tabella 1** is equal to 26 and are all printed in the following table:

| **#** | **# COPPIE VALORI** | | |
| --- | --- | --- | --- |
| **A, B** | **A, C** | **B, C** |
| 1 | A1,B1 | A1,C1 | B1, C1 |
| 2 | A1,B2 | A1,C2 | B1, C2 |
| 3 | A1,B3 | A2,C1 | B2, C1 |
| 4 | A2,B1 | A2,C2 | B2, C2 |
| 5 | A2,B2 | A3,C1 | B3, C1 |
| 6 | A2,B3 | A3,C2 | B3, C2 |
| 7 | A3,B1 | A4,C1 |  |
| 8 | A3,B2 | A4,C2 |  |
| 9 | A3,B3 |  |  |
| 10 | A4,B1 |  |  |
| 11 | A4,B2 |  |  |
| 12 | A4,B3 |  |  |
| **# COPPIE** | 12 | 8 | 6 |
| **TOTALE** | 12+8+6=26 | | |

**Tabella 6 –** Pairs of values of the variables A, B and C

Why you should consider a Test Set to cover wise-2? Not be enough to consider a Test Set with 1-wise coverage? Here we enter into a thorny issue, in which the opinions are varied, concordant and discordant.

Below is the “incipit” from the site <http://www.pairwise.org/> :

*“Pairwise (a.k.a. all-pairs) testing is an effective test case generation technique that is based on the observation that most faults are caused by interactions of at most two factors. Pairwise-generated test suites cover all combinations of two therefore are much smaller than exhaustive ones yet still very effective in finding defects.”*

We mention also the opinion of James Bach and Patrick J. Schroeder about the Pair Wise method: "Pairwise Testing: A Best Practice That Is not" from James Bach, Patrick J. Schroeder available from http://[www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf](http://www.testingeducation.org/wtst5/PairwisePNSQC2004.pdf) ):

“What do we know about the defect removal efficiency of pairwise testing? Not a great deal. Jones states that in the U.S., on average, the defect removal efficiency of our software processes is 85% [26]. This means that the combinations of all fault detection techniques, including reviews, inspections, walkthroughs, and various forms of testing remove 85% of the faults in software before it is released.

In a study performed by Wallace and Kuhn [27], 15 years of failure data from recalled medical devices is analyzed. They conclude that 98% of the failures could have been detected in testing if all pairs of parameters had been tested (they didn’t execute pairwise testing, they analyzed failure data and speculate about the type of testing that would have detected the defects). In this case, it appears as if adding pairwise testing to the current medical device testing processes could improve its defect removal efficiency to a "best in class" status, as determined by Jones [26].

On the other hand, Smith, et al. [28] present their experience with pairwise testing of the Remote Agent Experiment (RAX) software used to control NASA spacecraft. Their analysis indicates that pairwise testing detected 88% of the faults classified as correctness and convergence faults, but only 50% of the interface and engine faults. In this study, pairwise testing apparently needs to be augmented with other types of testing to improve the defect remove al efficiency, especially in the project context of a NASA spacecraft. Detecting only 50% of the interface and engine faults is well below the 85% U.S. average and presumably intolerable under NASA standards. The lesson here seems to be that one cannot blindly apply pairwise testing and expect high defect removal efficiency. Defect removal efficiency depends not only on the testing technique, but also on the characteristics of the software under test. As Mandl [4] has shown us, analyzing the software under test is an important step in determining if pairwise testing is appropriate; it is also an important step in determining what addition al testing technique should be used in a specific testing situation.”

**[4]** R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," Communication of the ACM, vol. 28, no. 10, pp. 1054-1058, 1985.

**[26]** Jones, Software Assessments, Benchmarks, and Best Practices. Boston, MA: Addison Wesley Longman, 2000.

**[27]** D. R. Wallace and D. R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," Int'l Jour. of Reliability, Quality and Safety Engineering, vol. 8, no. 4, pp. 351-371, 2001.

**[28]** B. Smith, M. S. Feather, and N. Muscettola, "Challenges and Methods in Testing the Remote Agent Planner," in Proc. 5th Int'l Conf. on Artificial Intelligence Planning and Scheduling (AIPS 2000), 2000, pp. 254-263

To clarify, we can say that Pairwise or 2-wise test method ensures that all combinations of pairs of values ​​of the variables are tested, "theoretically ensuring" the maximization of the anomalies found, with percentages ranging from 50% to 98% according to the studies. In fact, no test can ever guarantee a defined percentage removal of defects (which can **only** be calculated **ex post** for the **specific project**); let's say – trying to be realistic - that the Pairwise achieve a valid agreement between the number of tests to be performed and the anomalies detected, when the number of variables and their values is so high that it cannot be tested all the combinations (the so called all-wise testing or N-wise testing, where N is the number of variables we are playing with).

In the case of Test Set covering wise-2 level is very simple to know the **maximum number** of tests that provide coverage for all pairs of values of the variables. This value is equal to the number of pairs of values of the variables themselves. In our example, three variables A, B and C in **Tabella 1**, this number is 26 (calculated as described in **Tabella 6**). The real problem – still unsolved - is to discover the **minimum number** of tests that guarantees wise-2 coverage, although there are a variety of methods and algorithms that approximate this value for a problem with an arbitrary number of variables and values. Examples of tools that use these algorithms are:

1. Microsoft **P**airwise **I**ndependent **C**ombinatorial **T**esting tool (PICT), downloadable from <http://download.microsoft.com/download/f/5/5/f55484df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi>;
2. NIST Tools: <http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html>
3. James Bach AllPairs downloadable from <http://www.satisfice.com/tools.shtml> ;
4. Other tools here: <http://www.pairwise.org/tools.asp> .

# n-wise testing with n>2

It’s now easy to extend the concept of pairwise or 2-wise test to the generic case of n-wise, with n> 2. The generic Test Set provides n-wise coverage if it is able to cover all the n-tuples (3-tuples if n = 3, 4-tuples if n = 4 and so on). As in the Pairwise, is always possible to know the size of the maximum Test Set, equal to the number n-tuples of values of the variables, but there is no way to know - in the general case - the size of the Test Set Minimum guaranteeing coverage n -wise.

Using NIST Tools (ACTS) or PICT is possible to extract a Test Set that approximates as much as possible the minimum Test Set. It 'clear that, given a set of N variables, the maximum level of wise which you can have is equal to the number of variables. So, if we have 4 variables, a 4-wise test set coincides with all possible combinations, while a 5-wise Test Set or higher makes no sense.

# Wise crunch tools

The combinatorial testing techniques that we discussed in the preceding paragraphs are intended to solve a basic problem, which we have already discussed and we rephrase as follows:

*DIRECT COMBINATORIAL TEST PROBLEM: "A software system accepting N input variables, each of which can take on different values, find the Test Set with the smaller number of test cases that guarantee (at least) the coverage of all combinations (2-tuples) of all the values of the variables involved.”*

To solve this problem has been developed Pairwise technique and a large number of support tools. Once such a test set (the smallest possible) has been generated, you should run the test cases and detect (if present) all the defects comes up in the software under test.

There is also a second problem, maybe less "popular" compared to the previous, that is the following:

*REVERSE COMBINATORIAL TEST PROBLEM:"Given a Test Set for which you do not know the method of generation (if any), calculate what percentage of nwise-coverage the Test Set ensures, with nwise between 1 and the number variables in the Test Set ".*

An example is that in which tests are generated by automated tools for which you have a low or almost zero process control, or when the "test cases" are generated by the automatic flows that feed interfaces between different systems (think of a system that transmit accounting data from a system A to B); test data are - in general - excerpts from historical series on which you have no control.

For test scenarios in some way related to a combinatorial inverse problem, it’s not easy found support tools or such tools are not readily available. The only tool I found is NIST CCM, still in alpha-phase at the moment I’m writing this notes.

In the following we describe a set of tools called "*Wise crunch tools*" (executable under Windows, but -if needed– not difficult to port under Unix / Linux) allowing to extract the (quasi)minimal test set and calculate the coverage of a generic test set , using systematic algorithms calculation coverage, starting from all n-tuples of the variables’ values; such algorithms should be categorized "brute force algorithms" and should be used (on a normal supermarket-buy-PC) if the number of variables and values ​​is not too high.

## Installation procedure

The package "Wise crunch tools" consists of the following files and folders:

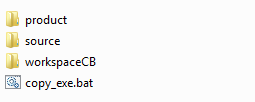


Figura 1 **– WiseCrunchTools high level file and folder**

The folder "***source***" keep the executable C++ sources files and it is not useful for the end user of the tools.

The folder "***workspaceCB***" keep workspace files used by the OpenSource C/C++ IDE called "Code::Blocks" (<http://www.codeblocks.org/> ) used during tools’ development; this folder is not interesting to the end user of the tools too. Obviously, you can use any C/C++ development environment if you have the interest to develop/modify tools.

The file "***copy\_exe.bat***" is a simple executable deploy utility, which copies the executables from the source folder *source\<nome tool>\bin\Release* of each tool in the*\product\bin* folder. Even in this file is not of interest for the end user of the tools.

The content of “product” folder is as follows:

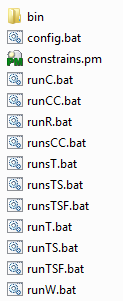


Figura 2 **– WiseCrunchTools “product” folder**

This folder contains the **main first level utilities** that will be described in detail in the following sections.

The content of “bin” folder is as follows:

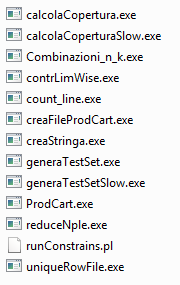


Figura 3 **– WiseCrunchTools “bin” folder**

This folder contains the **second level utility** that will be described in detail in the following sections.

The installation procedure is reduced to the following two simple steps:

1. Copy the contents of the "***product***" folder under the file system of the machine. We will assume in the following that the folder has been copied under ***C:\WiseCrunchTools\product.***
2. Add the following paths to system and/or user PATH variable, according with your user grant:
   1. *C:\WiseCrunchTools\product*
   2. *C:\WiseCrunchTools\product\bin*

In Windows, you can perform the operation from: *Control Panel → System → Advanced Settings*, press the "*Environment Variables*" button and then set the PATH system or user variable, according to the grant available. However, refer to the configuration manual of your Windows operating system just in case.

There are no special constraints if you choose to install the WiseCrunchTools in a folder whose name contains spaces, such as “*C:\My Documents\Wise Crunch Tools*”.

## Configuration procedure

The configuration procedure requires editing the file named "***config.bat***" in the folder "*product*." The file contents’ follows (ignoring the header comments):

REM Project's name

set "PROJECT=###### <NOME PROGETTO> ######"

REM Working directory (must end with \)

set "APP\_PATH=C:\Users\<utente>\Documents\<nome\_progetto>\"

REM Input file (must be present in the working directory)

set "INPUT\_FILE\_NAME=input.txt"

REM Separator \*\*MUST BE EQUAL\*\* to the separator used in the

REM input file to separate fields

set "SEP=;"

Set the following fields

* ***PROJECT*** field: enter the name of the project you will work on. It 'a simple mnemonic string and is useful as a reminder on what project you are working; enter any value in it.
* ***APP\_PATH*** field: enter a valid file system folder. On that folder all the data generated by tools you are going to use will be saved in this folder. The folder must end with the character "\" (slash).
  + <NOME VAR\_1 DI 1 CARATTERE>:VAL1\_1<SEP>VAL1\_2<SEP>VAL1\_3<SEP>…<SEP>VAL1\_N
  + <NOME VAR\_2 DI 1 CARATTERE>:VAL2\_1<SEP>VAL2\_2<SEP>VAL3<SEP>…<SEP>VAL2\_K
  + ….
  + <NOME VAR\_P DI 1 CARATTERE>:VALP\_1<SEP>VALP\_2<SEP>VALP<SEP>…<SEP>VALP\_T
* ***INPUT\_FILE\_NAME*** field: enter the name of a valid text file, containing variables and their values in the following format:
  + <NAME VAR\_1 DI 1 CHR>:VAL1\_1<SEP>VAL1\_2<SEP>VAL1\_3<SEP>…<SEP>VAL1\_N
  + <NAME VAR\_2 DI 1 CHR>:VAL2\_1<SEP>VAL2\_2<SEP>VAL3<SEP>…<SEP>VAL2\_K
  + ….
  + <NAME VAR\_P DI 1 CHR>:VALP\_1<SEP>VALP\_2<SEP>VALP<SEP>…<SEP>VALP\_T
* ***SEP*** field: the separator **must be the same** used in the input file (variable INPUT\_FILE\_NAME. The separator cannot be equal to ":" (colon) or "\*" (asterisk), that are reserved chars.

Variable names in the input file **must be a single character**; the best thing is to use the letters of the alphabet (upper or lower case and the numbers from 0 to 9).

An example of valid input file is the following:

**A:**A1;A2;A3;A4

**B:**B1;B2,B3

**C:**C1;C2;C3

In this example, the names of the variables (A, B and C) are separated from the list of values of the same from the character ":" (colon). The values of the separator is "," (comma) and must coincide with the variable set to the ***SEP*** file "***config.bat***".

## Overview of WiseCrunchTools

Tools in WiseCrunchTools product try to provide a support to the solution of both problems of combinatorial test previously stated.

WiseCrunchTools **do not intend to compete** with the existing tools aimed to solve the direct problem of combinatorial testing, such as Microsoft PICT, AllPair J. Bach, NIST or other several commercial and non-commercial tools already present on the market. These tools implement algorithms definitely more effective than WiseCrunchTools and *therefore should be favorite* – I repeat – to solve the direct problem.

Regarding the reverse problem of combinatorial tests do not exist, to my knowledge, tools on the market (except NIST CCM in alpha-test) and the WiseCrunchTools then attempt to provide a **very first** solution to the reverse problem, to be surely improved over time, when we will be better understood the logic and the rules that are behind combinatorial test and the minimal determination of test sets related to it.

We would like to remember that:

1. Solving the direct problem means determining the as small as possible test set with a level of WISE coverage agreed (usually WISE = 2) from the set of variables and values.
2. Solve the reverse problem means determining the level of coverage of a given test set with respect to a reference WISE level (also here usually is WISE = 2)

The tools should be categorized as follow:

1. *First level Tools*: batch DOS scripts providing an immediate response to standard scenarios that typically occur in test projects requiring combinatorial techniques.
2. *First level Tools*: executable (C++/Perl development language) more versatile giving response to problems that may be less common, but sometimes occurs in test projects requiring combinatorial techniques.

First level scripts were thought as the "wrapper" around the second level executables, in order to "simplify end user life" with a set of simple commands that allow to quickly get a number of “standard information”. The following table maps the two categories of tools and the tool with the kind of information they supply.

The second level executable called *uniqueRowFile.exe* is not invoked by the first level scripts and should be considered a special utility not used directly for the solution of the direct or reverse problem, but can be useful in specific advanced cases, remaining then only as a second-level utility.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| First Level | Second Level | | | | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| runW |  |  | X | X | X | X | X |  |  | X |  |  |  |
| runCC | X |  |  | X | X |  |  |  |  |  |  |  |  |
| runsCC |  | X |  | X | X |  |  |  |  |  |  |  |  |
| runT |  |  |  | X | X |  |  | X |  |  |  |  |  |
| runsT |  |  |  | X | X |  |  |  | X |  |  |  |  |
| runTS |  |  |  | X | X |  |  | X |  |  |  |  |  |
| runsTS |  |  |  | X | X |  |  |  | X |  |  |  |  |
| runTSF | X |  |  | X | X |  |  | X |  |  |  |  |  |
| runsTSF | X |  |  | X | X |  |  |  | X |  |  |  |  |
| runC |  |  |  |  |  |  |  |  |  |  |  | X |  |
| runR |  |  |  | X | X |  |  |  |  |  | X |  |  |

**Tabella 7 –First Level vs. second level tools mapping (wrapping map)**

Following is the list of second level executables:

1. calcolaCopertura.exe
2. calcolaCoperturaSlow.exe
3. Combinazioni\_n\_k.exe
4. contrLimWise.exe
5. count\_line.exe
6. creaFileProdCart.exe
7. creaStringa.exe
8. generaTestSet.exe
9. generaTestSetSlow.exe
10. ProdCart.exe
11. reduceNple.exe
12. runConstrains.pl
13. uniqueRowFile.exe

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tool | Level | Scenario | | | |  |  | Problem category |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 |
| runW | 1 | X |  |  |  |  |  | Reverse |
| runCC | 1 |  | X |  |  |  |  | Reverse |
| runsCC | 1 |  | X |  |  |  |  | Reverse |
| runT | 1 |  |  | X |  |  |  | Direct |
| runsT | 1 |  |  | X |  |  |  | Direct |
| runTS | 1 |  |  |  | X |  |  | Direct |
| runsTS | 1 |  |  |  | X |  |  | Direct |
| runTSF | 1 |  |  |  |  | X |  | Direct |
| runsTSF | 1 |  |  |  |  | X |  | Direct |
| runC | 1 |  |  |  |  |  | X | Direct/Reverse |
| runR | 1 |  |  | X |  |  |  | Reverse |
| calcolaCopertura | 2 |  | X |  |  |  |  | Reverse |
| calcolaCoperturaSlow | 2 |  | X |  |  |  |  | Reverse |
| Combinazioni\_n\_k | 2 |  |  |  |  |  |  | / |
| contrLimWise | 2 |  |  |  |  |  |  | / |
| count\_line | 2 |  |  |  |  |  |  | / |
| creaFileProdCart | 2 |  |  |  |  |  |  | / |
| creaStringa | 2 |  |  |  |  |  |  | / |
| generaTestSet | 2 |  |  | X | X | X |  | Direct |
| generaTestSetSlow | 2 |  |  | X | X | X |  | Direct |
| ProdCart | 2 | X |  |  |  |  |  | Inverso |
| reduceNple | 2 |  |  | X |  |  |  | Direct |
| runConstrains | 2 |  |  |  |  |  | X | Direct/Reverse |
| uniqueRowFile | 2 |  |  |  |  |  |  | / |

**Tabella 8 – Tools vs. Standard Scenari Mapping**

Following is the list of standard scenarios:

1. Calculating maximum set (all n-tuples)
2. Coverage Test Set calculation against a chosen level wise
3. Generate minimal Test Set with predefined wise coverage from all-combinations Test Set
4. Generate minimal Test Set with predefined wise coverage from a generic Test Set.
5. Generate minimal Test Set with predefined wise coverage from a generic Test Set excluding a set of already covered (from another test set) n-tuples.
6. Apply constrains to n-tuples.

**Tabella 8** shows how the tools deal with both types of problems; we want to emphasize, however, that if the main interest is to quickly produce Test Set covering a defined wise level, it is advisable use the tools already present in the market since long time. If you need to generate Test Set **from other test set** you already have, WiseCrunchTools should be the right choice (if the variables and data you have to crunch is not too high).

## Lexical and typographical conventions

In the following, the number of variables in the input file coincides with the maximum possible value of the coverage wise, will be indicated with WISE\_MAX.

We will often refer to a Minimum Test Set, considering it as a set containing the minimum number of test set as possible. Today is no algorithm guarantees that the Test Set extract is the absolute minimum, so speaking of minimum is not really correct. The right term would be almost-minimum, then the term "minimum" must be understood in this sense.

All the tools described below have a console help that can be accessed via the **/?** as usual for DOS commands. This option, being common to all tools, it will be not in the description of the specifications of the individual tool.

The parameters between braces **{...}** in **bold** are **REQUIRED**, while those between brackets *(...)* in *italics* are *OPTIONAL*.

## First level tools – batch DOS script

This chapter describes first level tools, easier to use, supporting the "standard scenario" described above

As a convention, the name of all the batch script follows the pattern:

* Prefix **run** common to all tools
* Letter **s** only for the so-called "slow" scripts, which optimize the memory usage increasing processing time.
* Number of uppercase letters **XX ... X** equal to the number of input script parameters

For example *runCC* is a script that takes two input parameters, while *runW* requires only one. The script *RunsCC* requires two input parameters and has a running time higher than *runCC*, but requires less system memory.

### Tool runW

It’s the first tool that is mandatory to run before all the other tools. It generate all the n-tuples corresponding to the value of the Wise past input (runW = run **W**ise)

#### Input:

1. **{WISE}**: Integer number between 1 and WISE\_MAX

#### Output:

1. File containing all the n-tuples corresponding to {WISE} passed in input, called ***out\_p\_{WISE}.txt*** in APP\_PATH folder
2. File with all the possible combinations of the input variables, called **out\_c\_{WISE}.txt** in APP\_PATH folder.

#### Example:

runW 2

Create all pairs of values ​​of the variables in the input file.

### Tool runCC and runsCC

Compute the input Test Set coverage respect the WISE passed as input (runCC = run **C**lculate **C**overage)

#### Input:

1. **{FILE TESTSET}:** The Test Set for which you want to calculate the coverage
2. **{WISE NUMBER}:** WISE value respect to which the coverage is to be calculated
3. *(-n)* Print n-tuples found in the file {FILE TESTSET}
4. *(-d)* Print details, consisting of the #row Test Set, #row N-tupla
5. *(-f)* Print frequency map of tuples: #row N-tupla, #occurrences n-tuple found
6. *(-r)* Print statistic (% coverage found)
7. *(-z)* Print the record in the Test Set with the number of n-tuples associated

#### Output:

1. The information as described by the options *(-n), (d), (f), (-r)* and *(-z)*. Note that if you don’tt pass at least one option, the command print nothing. The output is printed on the screen and can be redirected to a file with the usual operations "pipe" DOS (> or >>).

#### Prerequisites

1. The command *runW {WISE NUMBER}* must have been previously entered. Otherwise, a warning is printed with a request to execute it.

The tool ***runCC*** is faster but requires higher system memory. In the case where the system memory is not enough, the program is interrupted by a managed memory overflow. The tool ***runsCC*** (note the "**s**" in *run****s***which stands for "slow") requires less memory, but it takes more time, working mainly on file.

#### Example:

runCC test\_set.txt 2 –r -f

Prints out the statistics and the map of frequency for the test set contained in the file “test\_set.txt”.

### Tools runT and runsT

Get the minimal Test Set with guaranteed coverage equal to the Wise past in input (runT = run **T**est) extracting the test cases from the file of the WISE\_MAX-tuples (all combinations).

#### Input:

1. **{WISE}**: Integer number between 1 and WISE\_MAX

#### Output:

1. A file containing the test set with minimal {WISE} coverage guaranteed past in input, called ***test\_set\_generato\_{WISE\_MAX}\_{WISE}.txt*** in the execution folder.
2. A file containing the test set with minimal {WISE} coverage guaranteed past in input, called **clean\_*test\_set\_generato\_{WISE\_MAX}\_{WISE}.txt*** in the execution folder.

The difference between the two files is that ***test\_set\_generato\_{WISE\_MAX}\_{WISE}.txt*** shows the number of the extracted row from the all possible combinations WISE\_MAX-tuples file, while in **clean\_*test\_set\_generato\_{WISE\_MAX}\_{WISE}.txt*** this value is not reported.

#### Prerequisites

1. The command *runW {WISE NUMBER}* must have been previously entered. Otherwise, a warning is printed with a request to execute it.

#### Example:

runT 2

Extract the 2-wise Test Set coverage from the file that contains all the possible combinations.

Note that if {WISE} ≡ {WISE MAX} then the file *out\_p\_WISE\_MAX* is copied into the execution folder creating **only** the file **clean\_*test\_set\_generato\_{WISE\_MAX}\_{WISE\_MAX}.txt***. In this case is also printed a DOS error "File Not Found" caused by the condition that is not produced the file ***test\_set\_generato{WISE\_MAX}\_{WISE\_MAX}.txt***.

The tool ***runT*** is faster but requires higher system memory. In the case where the system memory is not enough, the program is interrupted by a managed memory overflow. The tool ***runsT*** (note the "**s**" in *run****s***which stands for "slow") requires less memory, but it takes more time, working mainly on file.

### Tools runTS and runsTS

Get the minimal Test Set with guaranteed coverage equal to the Wise past in input or equal to the coverage of the Test Set past in input if less than WISE, extracting the test cases from the file of the Test Set past in input (runTS = run **T**est **S**et)

#### Input:

1. **{FILE TESTSET BASE}** Input Test set from which extract the output Test Set
2. **{WISE}**: Integer number between 1 and WISE\_MAX

#### Output:

1. Minimal Test Set with guaranteed coverage equal to the Wise past in input or equal to the coverage of the input Test Set {FILE TESTSET BASE} if less than WISE, called ***test\_set\_generato\_{WISE}.txt*** in the execution folder.
2. Minimal Test Set with guaranteed coverage equal to the Wise past in input or equal to the coverage of the input Test Set {FILE TESTSET BASE} if less than WISE, called ***clean\_test\_set\_generato\_{WISE}.txt*** in the execution folder.

The difference between the two files is that ***test\_set\_generato\_{WISE}.txt*** shows the number of the extracted row from the input Test Set {FILE TESTSET BASE}, while in **clean\_*test\_set\_generato\_{WISE}.txt*** this value is not reported.

#### Prerequisites

1. The command *runW {WISE}* must have been previously entered. Otherwise, a warning is printed with a request to execute it.

#### Example:

runTS base\_testset.txt 2

Extract 2-wise Test Set coverage (or coverage equal to the coverage of the base\_testset.txt Test Set past in input if less than WISE) from the input file base\_testset.txt.

To remark that if the file{FILE TESTSET BASE} past as input does not guarantee coverage to {WISE} level passed as input, the output files will coincide with the input file and will have - of course - the same coverage of the input file {FILE TESTSET BASE}.

The tool ***runTS*** is faster but requires higher system memory. In the case where the system memory is not enough, the program is interrupted by a managed memory overflow. The tool ***runsTS*** (note the "**s**" in *run****s***which stands for "slow") requires less memory, but it takes more time, working mainly on file.

### Tools runTSF and runsTSF

Get the minimal Test Set with guaranteed coverage equal to the Wise past in input or equal to the coverage of the Test Set past in input if less than WISE, extracting the test cases from the file of the Test Set past in input, excluding n-tuples already covered by the Partial Test Set input file (runTSF = run **T**est **S**et **F**orbidden).

#### Input:

1. **{FILE TESTSET BASE}** Input Test set from which extract the output Test Set
2. **{FILE TESTSET PARZIALE}** Partial Test Set with valid test cases, whose WISE-tuples are to be excluded from the output to be generated.
3. **{WISE}**: Integer number between 1 and WISE\_MAX

#### Output:

1. Minimal Test Set with guaranteed coverage equal to the Wise past in input “minus” the tuples in {FILE TESTSET PARZIALE}or equal to the coverage of the input Test Set {FILE TESTSET BASE} if less than WISE “minus” the tuples in {FILE TESTSET PARZIALE}, called ***delta\_test\_set\_generato\_{WISE}.txt*** in the execution folder.
2. Minimal Test Set with guaranteed coverage equal to the Wise past in input “minus” the tuples in {FILE TESTSET PARZIALE}or equal to the coverage of the input Test Set {FILE TESTSET BASE} if less than WISE “minus” the tuples in {FILE TESTSET PARZIALE}, called ***delta\_clean\_test\_set\_generato\_{WISE}.txt*** in the execution folder.

The difference between the two files is that ***delta\_test\_set\_generato\_{WISE}.txt*** shows the number of the extracted row from the input Test Set {FILE TESTSET BASE}, while in ***delta\_*clean\_*test\_set\_generato\_{WISE}.txt*** this value is not reported.

To remark that the generated Test Set file is only the **delta** of test cases than those contained in {FILE TESTSET PARZIALE}; then the generated file will never have a coverage equal to WISE past as input (or equal to the coverage of the input Test Set {FILE TESTSET BASE} if less than WISE). This coverage is only guaranteed from the file "sum" of delta\_\*\_*test\_set\_generato\_{WISE}.txt*with {FILE TESTSET PARZIALE}

#### Prerequisites

1. The command *runW {WISE}* must have been previously entered. Otherwise, a warning is printed with a request to execute it.

#### Example:

runTSF base\_testset.txt parziale.txt 2

Suppose the file base\_testset.txt has a WISE coverage = 3 respect the variables he manages. The above command extract 2-wise coverage Test Set without the tuples in parziale.txt, from the input file base\_testset.txt.

The tool ***runTSF*** is faster but requires higher system memory. In the case where the system memory is not enough, the program is interrupted by a managed memory overflow. The tool ***runsTSF*** (note the "**s**" in *run****s***which stands for "slow") requires less memory, but it takes more time, working mainly on file.

### Tool runR

Extracts a **Non Minimal** Test Set but still smaller than the Maximum Test Set with guaranteed coverage equal to the Wise passed as input (runR = run **R**educe).

#### Input:

1. **{WISE}**: Integer number between 1 and WISE\_MAX

#### Output:

1. Test Set smaller than Maximum Test Set, but not minimal, with coverage guaranteed equal to past input WISE. Values with \* are the "don’t care". Iinstead of "\*", you can take any value of the variable, without thereby affecting the coverage WISE level. The output is printed on the screen and can be redirected to a file with the usual operations "pipe" DOS (> oppure >>)

The number of test cases in the output Test Set depend form the order of the n-tuples in *out\_p\_{WISE}.txt* file that is passed as input to *runR.bat*. Surely there is a sort order of this file file to which the output Test Set contains a minimum number of Test Cases, but find this sorting is not feasible from a computational point of view, as too onerous.

#### Prerequisites

1. The command *runW {WISE}* must have been previously entered. Otherwise, a warning is printed with a request to execute it.

#### Example:

runR 2

Create test set with wise-2 coverage that is not reduced to the minimum, but with a number of Test Cases less than the maximum. The numbers of test cases depends on the ordering of the n-tuple in the file *out\_p\_2.txt* in APP\_PATH.

### Tool runC

Apply constrains to n-tuple file (or Test Set file) past as input.

#### Input:

1. **{FILE }:** The Test Set or n-tuples file to which you want to apply constraints programmed into the file *constrains.pm* (Perl module)

#### Output:

1. On the standard outpu are printed valid n-tuples or test cases; while on the standard error are printed invalid n-tuples or test cases. The output is printed on the standard output (usually the screen) and the standard error (usually coinciding with the standard output) and can be redirected to a file with the usual operations "pipe" DOS (> or >>).

#### Prerequisites

1. No one. The tool is not directly linked to other tools.

#### Example:

runC test\_set.txt 1>validi.txt 2>non\_validi.txt

Apply constraints programmed into the file *constrains.pm* (Perl language) and prints the valid records in the file "*validi.txt*" (records that **meet** the constraints) and invalid records in the file "*non\_validi.txt*" (records that do **not meet** the constraints).

The file "constrains.pm" containing the filters is a module written in Perl, in which you can write the conditions that allow you to accept or discard the records in the file passed as input. The variable to be considered is the vector **$r** containing the values of the variables of the record read. In practice, for each line read, the following association is executed:

$r->[0]: current value of the first variable (A) on the line currently read

$r->[1]: current value of the second variable (B) on the line currently read

…

$r->[K]: current value of the (K-1)-th variable (J) on the line currently read

Each filter must return a value. The possible values are 0 or 1:

* 1: the row is considered **not valid**
* 0: the row is considered **valid**

For example

if($r->[0] eq "A1" && $r->[1] eq "B3"){

return 1;

};

Each line of the input file is read and the condition is verified: if the value of the first variable A is "A1" and the value of the second variable B is "B3", line is discarded. In the current release of the product has not been implemented a mechanism to vary the values of the variables in the line of the file, in other words, a condition of the type:

if($r->[0] eq "A1" && $r->[1] eq "B3"){

$r->[1]="B2"

return 0;

};

**would not change** the value of the second variable from "B3" to "B2"; then retuning 0, the line that should be excluded, would instead - wrongly - be taken as valid.

In other word, the filter validates or not validates the values, but does not change the content of the same.

## Second level Tools – executable

This chapter describes the second level tools, a little more hard to use but more versatile, which in addition to supporting the "standard"cscenarios described above, may be useful to experienced user to manage more complex scenarios.

### Executable calcolacopertura.exe and calcolaCoperturaSlow.exe

Performs the coverage calculation of the input Test Set respect the input WISE.

#### Input:

1. –**w: {WISE FILE NAME}** File name with all the {WISE}-tuples , file *out\_p\_{WISE}.txt*
2. **–t: {FILE TESTSET}:** The Test Set for which you want to calculate the coverage
3. **–o: {WISE NUMBER}:** The reference WISE value
4. *(-s)* Field file separator
5. *(-n)* Print n-tuples found in the file {FILE TESTSET}
6. *(-d)* Print details, consisting of the #row Test Set, #row N-tupla
7. *(-f)* Print frequency map of tuples: #row N-tupla, #occurrences n-tuple found
8. *(-r)* Print statistic (% coverage found)
9. *(-z)* Print the record in the Test Set with the number of n-tuples associated

#### Output:

1. The information as described by the options *(-n), (d), (f), (-r)* and *(-z)*. Note that if you don’tt pass at least one option, the command print nothing. The output is printed on the screen and can be redirected to a file with the usual operations "pipe" DOS (> or >>).

The program ***calcolaCopertura*** is faster but requires higher system memory. In the case where the system memory is not enough, the program is interrupted by a managed memory overflow. The proram ***calcolaCoperturaSlow*** requires less memory, but it takes more time, working mainly on file.

#### Example:

calcolaCopertura –w: out\_p\_2.txt –t: test\_set.txt –o: 2 –r -f

Prints out the statistics and the frequency map of the test set contained in the file “test\_set.txt” with respect to wise level 2.

### Executable Combinazioni\_n\_k

Extracts all K by K combinations of a string of length N past as input.

#### Input:

1. –**s: {STRINGA}** Input string of with all distinct characters
2. **–k: {k}:** Value of the length of the chunk of string (K by K combinations)

#### Output:

1. All combinations of the string K by K. The output is printed on the screen and can be redirected to a file with the usual operations "pipe" DOS (> or >>)

#### Example:

Combinazioni\_n\_k –s: ABCDE –k: 2

Returns:

A B

A C

A D

B C

B D

C D

These are the combinations of the string ABCDE taken 2 by 2. Note that if the string contains duplicate characters (eg ABCBCD, which contains the character "B" two times), the program returns an error.

### Executable generaTestSet.exe and generaTestSetSlow.exe

Get the minimal Test Set with guaranteed coverage equal to the Wise past in input or equal to the coverage of the Test Set past in input if less than WISE, extracting the test cases from the file of the Test Set past in input, excluding n-tuples already covered by the Partial Test Set input file.

#### Input:

1. –**w: {WISE FILE NAME}** File name with all the {WISE}-tuples , file *out\_p\_{WISE}.txt*
2. **–t: {FILE TESTSET}:** The Test Set for which you want to calculate the coverage
3. **–o: {WISE NUMBER}:** The reference WISE value
4. *(-s)* Field file separator
5. *-f:(FROM FILE)*  File with list of the rows of the file *out\_p\_WISE.txt* (tuples) to be excluded from the output to be generated.

#### Output:

1. Test set with the coverage required as from input parameters. The output is printed on the screen and can be redirected to a file with the usual “pipe” DOS operations (> or >>).

The program ***generaTestSet*** is faster but requires higher system memory. In the case where the system memory is not enough, the program is interrupted by a managed memory overflow. The proram ***generaTestSetSlow*** requires less memory, but it takes more time, working mainly on file.

#### Example:

generaTestSet –w: out\_p\_2.txt –t: test\_set.txt –o: 2 –f: nple.txt

Generate a Test Set from the file test\_set.txt with coverage equal to 2 or equal to test\_set.txt file coverage if less than 2, extracting the test cases from the n-tuples contained in the file out\_p\_2.txt from which excludes the n-tuples contained in the file nple.txt file. This file is a list of the form:

12

14

15

26

23

…

The list is composed by the row numbers of the out\_p\_2.txt file containing all the tuples with wise coverage = 2; the n-tuples contained in the file nple.txt will be excluded when generating the output Test Set. The file with the n-tuples to be excluded can be generated with the executable *calcolaCopertura.exe* (or *calcolaCoperturaSlow.exe*) with only the option *(-n)* applied to the test set from which they want "extract" the n-tuples to be excluded.

### Executable ProdCart.exe

Generate all possible combinations of the values of variables as defined in the input file

#### Input:

1. –**i: {INPUT FILE NAME}** Input file in which each line has all the variables values separated by(SEP)
2. *–s: (SEP)***:** Separator of the values of the variables. The default is ";" (semicolon).
3. *(-c)* Output comments during generation of tuples.

#### Output:

1. A file with all the possible combinations of input variables. The output is printed on the screen and can be redirected to a file with the usual operations "pipe" DOS (> or >>)

The program ***ProdCart*** is generally much faster than the previous ones, but in the case where the system memory is not sufficient (ie, when the number of combinations is very high), exits with a managed memory overflow.

#### Example:

ProdCart –i: input.txt

The command outputs all the possible combinations of variables. If, for example, the file input.txt is as following:

A1;A2;A3;A4

B1;B2;B3

C1;C2

The output including all combinations will be:

A1;B1;C1

A1;B1;C2

A1;B2;C1

A1;B2;C2

A1;B3;C1

A1;B3;C2

A2;B1;C1

A2;B1;C2

A2;B2;C1

A2;B2;C2

A2;B3;C1

A2;B3;C2

A3;B1;C1

A3;B1;C2

A3;B2;C1

A3;B2;C2

A3;B3;C1

A3;B3;C2

A4;B1;C1

A4;B1;C2

A4;B2;C1

A4;B2;C2

A4;B3;C1

A4;B3;C2

ie, all combinations of the values ​​of all the three variables A, B and C, one for each line of the input file. Note that the lines in the input file ProdCart are not labeled as in the case of the input file of the batch tool *runW* (see the section “Configura”). In that case the input file would have to report to the head of each line the variable name (composed of a single letter) followed by ":", ie:

**A:**A1;A2;A3;A4

**B:**B1;B2;B3

**C:**C1;C2

**But** this file format **is not correct** for the executable ProdCart.

### Executable reduceNple.exe

“Squeeze” as much as possible n-tuples contained in the input file, replacing the values"\*" with specific values of the variables and thus creating a Test Set from the file of n-tuples, while not the Test Set minimum, it is reduced compared to the Test Set Maximum (coincident with all n-tuples). The number of records depends from the sorting of the n-tuples input file, in a not know way. For sure there is a sorting of the files row to which the Test Set outputted contains a minimum number of Test Cases with guaranteed WISE-coverage, but find this sort is not feasible from a computational point of view, as too onerous.

#### Input:

1. **–n: {NPLE FILE}**: File with n-tuple to reduce
2. *–s: (SEP)***:** Separator of the values ​​of the variables. Default is ";" (semicolon).

#### Output:

1. Test Set smaller than Maximum Test Set, but not minimal, with coverage guaranteed equal to past input n-tuple WISE file. Values ​​with \* are the "don’t care". Instead of "\*", you can take any value of the variable, without thereby affecting the coverage WISE level. The output is printed on the screen and can be redirected to a file with the usual operations "pipe" DOS (> or >>)

#### Example:

reduceNple –n: out\_p\_1.txt

Supponendo che il file out\_p\_1.txt sia il seguente e che garantisca una copertura 1 dei valori delle variabili A, B e C:

Assuming that the file out\_p\_1.txt is the following and which guarantees a cover 1 of the values ​​of the variables A, B and C:

A1;\*;\*

A2;\*;\*

A3;\*;\*

A4;\*;\*

\*;B1;\*

\*;B2;\*

\*;B3;\*

\*;\*;C1

\*;\*;C2

The reduced test set is the following:

A1;B1;C1

A2;B2;C2

A3;B3;\*

A4;\*;\*

In this case the output produced also coincides with the minimum Test Set, although in the general case when wise> 1 this is not true.

### Utility executable

The executables briefly described below does not provide direct support to the generation and/or operation on the Test Sets, but are predominantly used by DOS batch tools to perform secondary operations that it is impossible or - at least - very complex to do directly from DOS. These utilities may also be of some usefulness, even if they are not to be considered "tout cours" test tools. We document them briefly below for completeness, noting that simply launching the command without passing parameters, is printed (for the most significant ones) a help on how to use the command.

* ***contrLimWise.exe***: performs the control of the minimum and maximum limits of WISE values within the batch. If launched, it returns **"ko"** and - taken by itself - it is of no practical use and does not return a help on how to use them.
* **count\_line.exe**: count the lines in the file passed as input and prints the value on the screen. If input has 10 records, then count\_line Input.txt returns 10. If launched without input parameter returns a help on how to use it.
* **creaFileProdCart.exe**: creates a file in a format acceptable by *ProdCart.exe*, taking care only of variables on which the combinations must be calculated and replacing the other values with "\*". Taken by itself, is of no practical use and launched with no parameters returns a help on how use it.
* **creaStringa.exe**: creates a string containing the names (one length character) of the variables in the input file as described in “Configura”. Taken by itself, is of no practical use and launched with no parameters returns a help on how to use it.
* **runConstrains.pl:** is not a real executable, but a Perl script used by the tool runC to apply constraints to Test Set or to n-tuples file. Not to be used directly, unless it is well aware of what you are doing.
* **uniqueRowFile.exe**: filters out duplicate lines from the file passed as input and print out the uniques ones to the standard output. ***Do not change the input file***. Launched without parameters, it returns a help on how to use it and can be of practical use if you have files with a lot of duplicated lines to "clean". This executable is not used within the first level script and should be considered a self-consistent utility that can be useful in specific cases, thus remaining only a second level utility.

# Document’s revision

| **Version** | **Description** |
| --- | --- |
| **01** | First release on 20 february 2014 |