

**□Mika**

**User Manual**

**Version 1.1.2**

**Midoan Software Engineering Solutions Ltd.**

Copyright © 2009 Midoan Software Engineering Solutions Ltd.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by Midoan Software Engineering Solutions Ltd.

**Abstract:** Mika is an automatic test data generation tool for code written in a subset of Ada 83, Ada 95 or Ada 2005. Mika uses genetic algorithms in its attempt to generate inter-subprograms tests that should, by construction, exercise, during execution, the maximum possible number of branches, or

decisions, in the code under test.

**Keywords:** automatic test data generation, unit testing, integration testing, Ada, Spark Ada

## Table of Contents

<a href="#">1.Introduction.....</a>	<a href="#">3</a>
<a href="#">2.Requirements.....</a>	<a href="#">3</a>
<a href="#">3.Installation.....</a>	<a href="#">3</a>
<a href="#">4.Ada Subset.....</a>	<a href="#">4</a>
<a href="#">5.Limitations.....</a>	<a href="#">5</a>
<a href="#">i.Complexity Limitations .....</a>	<a href="#">5</a>
<a href="#">ii.Accuracy Limitations.....</a>	<a href="#">5</a>
<a href="#">iii.Miscellaneous Limitations.....</a>	<a href="#">6</a>
<a href="#">6.The GUI Front End.....</a>	<a href="#">6</a>
<a href="#">i.GUI Area 1: Drive and Folder Choosing.....</a>	<a href="#">6</a>
<a href="#">ii.GUI Area 2: Package, Subprogram and Test File Choosing.....</a>	<a href="#">6</a>
<a href="#">iii.GUI Area 3: Parsing.....</a>	<a href="#">7</a>
<a href="#">iv.GUI Area 4: Test Data Generation.....</a>	<a href="#">7</a>
<a href="#">v.GUI Area 5: Progress.....</a>	<a href="#">8</a>
<a href="#">vi.GUI Area 6: File View.....</a>	<a href="#">8</a>
<a href="#">7.The Tools Back End.....</a>	<a href="#">8</a>
<a href="#">i.The mika_ada_parser Parser Tool.....</a>	<a href="#">8</a>
<a href="#">ii.The mika_ada_generator the Test Data Generator.....</a>	<a href="#">9</a>
<a href="#">8.Examples.....</a>	<a href="#">10</a>
<a href="#">9.Feedback and Bug Reports.....</a>	<a href="#">10</a>

## 1. Introduction

Mika is a new generation testing tool; in fact we believe that it is the first tool of its kind for industrial code written in a professional programming language.

Mika generates tests that will achieve, when executed, the highest possible branch or decision coverage of your code.

Mika does not generate a large number of tests, nor any random tests, to achieve its aim: each test is carefully constructed to exercise a previously uncovered decision or branch in the code under test.

Mika offers, for the first time, the possibility to automatically generate test data to achieve maximum possible coverage of the source code under test: the only necessary input is the original source code. There are no annotations, no testing scripts to write.

Mika can generate under ten minutes a small set of tests from subprograms that may lead to the execution of 100 000s of line of code. The tests generated will be aimed at achieving the highest possible code coverage of the subprogram under test and of all the subprogram it itself calls.

Mika works for a substantial subset of Ada that largely encompasses the SPARK Ada subset: if you have SPARK Ada like code (Mika does not rely on annotations) Mika should be able to automatically generate targeted test data using the source code as-is.

Mika has been designed to leave your original code and the containing folders untouched: your code will not be modified in any way during the test data generation process.

Mika has been designed to be as simple as possible to use: pick the file containing the Ada subprogram you wish to generate tests for, pick the subprogram and click the generate tests button. You should be able to evaluate the suitability of Mika for your own circumstances in minutes.

Mika does not aim to replace traditional well established coverage testing tools: its main functionality is the automatic creation of purpose-built test data that should, by construction, achieve the highest possible level of coverage of your source code. It replaces a manual process. These tests can serve as input into traditional tools to certify the level of testing coverage achieved.

Mika replaces the tedious, expensive, time-consuming, error prone and usually incomplete, process of manual test data creation.

Mika generates tests that exercise called subprograms as thoroughly as the caller subprogram: tests are not just unit tests; they can be used for integration testing purposes.

Mika is not a static analysis tool: no false positive are ever generated. The tests it produces are complete and directly executable.

Mika provides, at no extra cost, the predicted outputs that will be generated by the code under test given the input tests generated: you can validate the behaviour of your code without having to actually execute it if you wish.

Mika provides, with very little additional overheads, an automatically generated test driver that uses

the tests automatically generated. This test driver can be compiled and run automatically.

Automatic testing for Ada has arrived.

## 2. Requirements

- Microsoft Windows XP (Mika is untested for other Operating Systems) with .NET v2.0 or above installed;
- A working installation of the GNAT Ada compiler (e.g. GNAT Pro, GNAT GPL, MinGW's GNAT).

## 3. Installation

Unzip the `Setup_Mika.zip` file (which can be downloaded from [http://midoan.com/download/current/Setup\\_Mika.zip](http://midoan.com/download/current/Setup_Mika.zip)) anywhere and double click on `setup.exe`; install Mika in a directory of your choice. Follow Mika's installation process and first use configuration dialogs.

## 4. Ada Subset

General remarks that should be kept in mind when considering the subset handled by Mika:

1. There are no theoretical limits on the type of constructs that Mika could handle: future versions of Mika will tackle larger Ada subset;
2. Typically, if your code contains a construct that Mika does not handle, a warning will be issued and the construct will be ignored: the test data generation process will continue. The tests generated may still achieve the level of coverage predicted, the predicted behaviour may still be correct: these should always be confirmed using a third party code coverage tool (possibly using Mika's automatically generated test driver as input);

The Ada subset that Mika is able to tackle is described below:

1. The subset is based on Ada 2005;
2. Ada tasks are not handled;
3. Access types are not handled;
4. Exceptions are not handled;
5. Tagged types are not handled;
6. Generic Packages are not handled;
7. Address clauses are not handled;
8. Machine code insertions are not handled;

9. Discriminated record types are not handled;
10. Records with variant parts are not handled;
11. Exit statements with named loop are not handled;
12. Static Null ranges are not handled;
13. Bitwise operators are not handled;
14. Use of predefined libraries subprogram is limited.

Handled features of note:

1. Mika does not put any restrictions on scope, visibility, renaming and overloading rules for the code under test;
2. Unconstrained arrays are handled;
3. Recursion is handled;
4. Default expressions of record components, subprogram parameters are handled;
5. The Ada subset handled by Mika fully subsumes the SPARK Ada subset (but does require any annotations).

## **5. Limitations**

Because of its nature, Mika may not be able to handle code of arbitrary complexity within a reasonable time with the memory at its disposal. Also, Mika may not be able to accurately handle code that uses some constructs.

### **i. Complexity Limitations**

The complexity of the control flow graph of the code under test has a direct impact on Mika's ability to generate tests within a reasonable time and within the memory at its disposal. While Mika can handle very large amounts of linear code (100 000s lines), subprogram calls and bounded loops on simple data structures without problems, the following aspects of the code under test will have a negative impact on Mika's run-time and/or make Mika reach its memory limitations:

1. Input dependent loop;
2. Input dependent recursion;
3. Input dependent unconstrained arrays;
4. Input dependent discrete ranges of slices;

5. Iterations over large data structures.

It is difficult to offer a more precise picture of Mika complexity limitations.

Mika has been designed however to produce results within the reasonable time of the order of minutes. For example, it is not uncommon for the compilation time of the automatically generated test driver to be much larger than the test data generation process time itself.

## **ii. Accuracy Limitations**

The following accuracy limitations are part of Mika:

1. Large integers are not properly handled;
2. Very large floating point numbers are not properly handled;
3. Floating point numbers representation is approximate;
4. Input dependent non-static types are not accurately handled;
5. Input dependent, non static, default field values in record component are not accurately handled;
6. Input dependent loop ranges are not accurately handled;
7. Modular types are not accurately handled;
8. Array slicing is not always fully accurately handled;
9. Boolean operators on arrays of Booleans are not accurately handled;
10. Overloading of Boolean operators with non Boolean returning operators are not accurately handled;
11. Intrinsic subprograms are not all accurately handled;

These accuracy limitations imply that, on occasions, the predicted coverage by Mika will not be achieved during actual execution or that the expected outcome will differ.

## **iii. Miscellaneous Limitations**

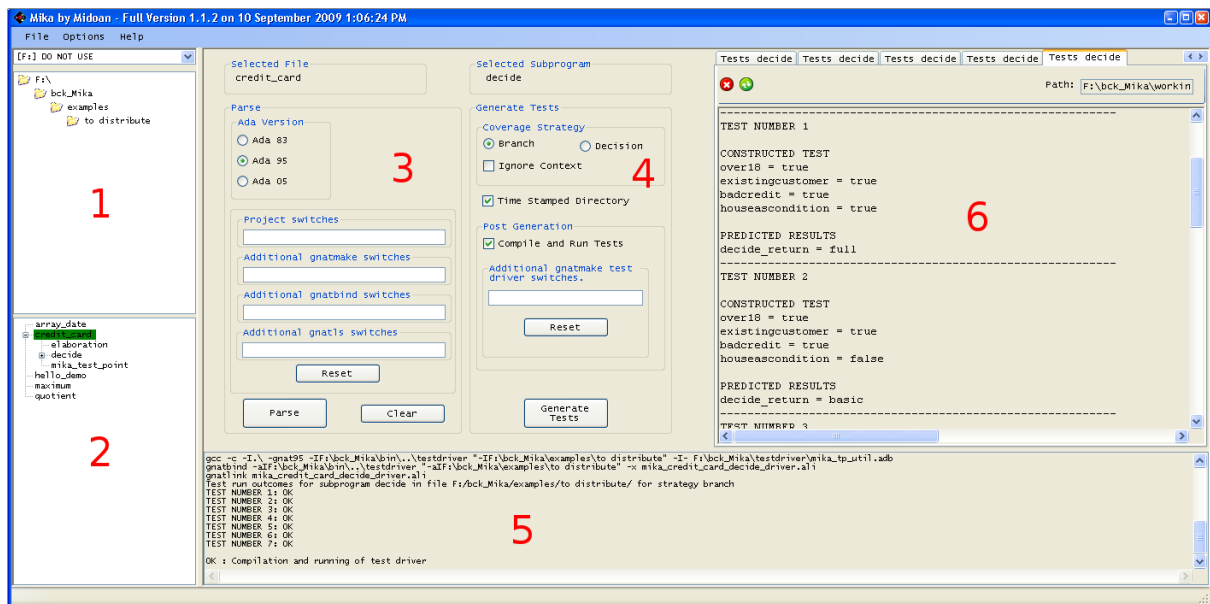
Currently include:

1. Subprograms local to another subprogram cannot be directly tested.

## **6. The GUI Front End**

Mika can be used via the simple GUI front end or, as detailed in the next section, via the command line for easier integration with your favourite IDE)

The GUI is divided into 6 main areas;



### i. GUI Area 1: Drive and Folder Choosing

This where you may pick the folder containing the code you wish to test. Mika searches, in the background, the contents of the chosen folder for Ada source code containing packages (.ads and .adb files) and displays the results in the area 2.

### ii. GUI Area 2: Package, Subprogram and Test File Choosing

The names of the top most packages contained in the chosen folder are displayed. If a package has never been parsed, its background appears transparent; if it has already been parsed but is out of date (i.e. it probably needs re-parsing unless the changes made on the source code have no semantics consequences) its background appears blue; finally if the package is has already been parsed and is up to date, its background appears green. In any case, an already parsed subprogram can be expanded to reveal its subprograms (and the elaboration only test data generation option). Previous test data generations outcomes for a given subprogram can be further expanded and viewed. Once a package file has been selected its containing file is displayed in area 6;

### iii. GUI Area 3: Parsing

A package that has not yet been parsed needs parsing. Note that the parser will re-compile your files if they are out of date, prior to its own processing. Default switches for the compiler are displayed and can be modified manually. For example, you can specify the path of the original gnat.adc file by adding -gnatec="C:\foo\gnat.adc" to the gnatmake switches box on the GUI (do not forget to add it also to the test driver switches box if you wish to compile the auto generated test driver). The Reset button uses the settings set via the Options->Set Default Switches... to reset the parse switches for the current package under test. The Clear button allows erasing of previous parsing outcomes for the package under tests. The Parse button parses the file where the chosen package is to be found. Progress is shown in the area 5. The path to the compiler can be set via Options->Set Working Directory...

#### iv. GUI Area 4: Test Data Generation

Once a subprogram has been chosen, the test data generation process can be customised. Branch or Decision coverage can be chosen. The calling context of the subprogram can also be ignored by ticking the Ignore Context option (this is usually what is meant by unit testing) or taken into account. If the context is taken into account, the tests generated will be based according to the outcome of the elaboration phase of the package containing the subprogram under test. In other words, the elaboration context will be taken into account. These lead to more realistic tests, but may also lead to a lower level of coverage. On the other hand, ignoring the elaboration context allows the generation of test data that overwrites initialisations made at elaboration time. The test may be less realistic but typically will achieve a higher level of coverage. If the Time Stamped Directory option is chosen, the test files are created in a new directory within the working directory whose name is time stamped. The current working directory can be changed via Options->Set Working Directory... The time stamped directory is thus unique and takes the form:

```
<subprogram_name>_year_month_day_hour_minutes_seconds
```

, otherwise the directory is just <subprogram\_name> and previous tests will be lost. Note that if the subprogram under test is an operator the <subprogram\_name> is known internally as string<\_ascii\_code>\* (E.g. "\*" will be known as string\_42, and "and" will be known as string\_97\_110\_100).

The Compile and Run Tests option compiles and runs the automatically generated test driver. To avail of this option users must manually (Mika is never allowed to automatically change user's code) insert test points in the specification and body of the packages under test. These tests point must be of the form:

```
procedure Mika_Test_Point (Test_number : in Integer);
```

in the specification part. And:

```
procedure Mika_Test_Point (Test_number : in Integer) is separate;
```

in the body part. A file named packageName-mika\_test\_point.adb must be added in the source folder (Mika never compromises the integrity of the original source folder: it always works in its own working directory, never writing anything, nor modifying anything in the original source folder). This test point file must be of the form:

```
separate (packageName)
```

```
procedure Mika_Test_Point (Test_number : in Integer) is
```

```
begin
```

```
    null;
```

```
end Mika_Test_Point;
```

where packageName should be replaced by the actual package name. Test points only need to be added if the user wishes to avail of the automatically generated test driver. If the elaboration context

is ignored, many more than just the package containing the subprogram under test may require the manual insertion of a test point.

In addition, the automatic generation of the test driver may need some help to produce Ada code that actually compiles. Users can add context clauses to the automatically generated test points by preceding the first lines of their own corresponding test point with `--MIKA`. E.g. saving:

```
--MIKA with System; use System;
separate (Hardware)
procedure Mika_Test_Point(Test_number : in Integer) is
begin
    null;
end Mika_Test_Point;
```

in `hardware-mika_test_point.adb` in the current working directory will ensure that the automatically generated test driver will systematically contain the context: `with System; use System;`. Basically the rest of the line following `--MIKA` will be automatically added at the start of the corresponding automatically generated test point.

Actual test outcomes are automatically compared against expected test outcomes. Compilation switches for the test driver can be manually changed or reset. The `Generate Tests` button generates the tests according to the user settings. Progress is shown in the area 5. Actual tests are displayed in area 6.

## v. GUI Area 5: Progress

Warnings and errors: these are displayed in this area.

## vi. GUI Area 6: File View

This area contains a simple file viewer to view (but not edit, so as to minimise the risk of corrupting production code) source code and generated tests. The generated tests are contained in a file named `<file_name_no_extension>_mika_tests.txt` in the current working directory.

## 7. The Tools Back End

Mika can also be used directly via the command line. This does not provide new functionalities as the GUI front end offers all the functionalities of the back end tools, but allows users to integrate Mika in their favourite environment (e.g. GPS).

There are two tools provided, `mika_ada_parser` the parser tool and `mika_ada_generator`, the test data generator tool.

### i. The `mika_ada_parser` Parser Tool

The parser tool, `mika_ada_parser`, parses the indicated file, and all dependent files, to produce

an output suitable for the subsequent test data generation phase handled by the `mika_ada_generator` tool.

Note that if the file under test, or any of its dependences, is not up to date, `mika_ada_parser` will compile it prior to the parsing phase proper. You can avoid this behaviour by always ensuring that the file under test and all its dependences are up to date prior to calling `mika_ada_parser`. As explained below, you can control this behaviour by providing a GNAT project file via the `-e` switch and a target working directory via the `-w` switch.

The basic syntax (note that the double quotes are actually necessary on the command line) is:

```
mika_ada_parser -M"<full_path_of_install_dir>" (-gnat83|-gnat95|-gnat05) [optional_switches] <file_name_no_extension>
```

the order of the switches is not significant.

`-M"<full_path_of_install_dir>"` is compulsory and must provide the full path to the `bin` directory of Mika file e.g. `-M"C:\Program Files\Midoan Software Engineering Solutions\Mika\bin"`;

one of the Ada version switch (i.e. `-gnat83` or `-gnat95` or `-gnat05`) must be provided (even if a GNAT project file is used) to indicate the Ada standard of the file under test;

the last argument must be the name of the file under test without extension e.g. :  
`oilandgasmonitoring-gasflowcalibrationcheck`.

Optional Switches:

`-f"<full_path_to_gnat_bin>"` provides the full path to `bin` directory of the gnat version you which to use e.g. `-f"F:\GNAT\2009\bin"`. If it is not specified, the gnat version reachable via your `PATH` environment variable is used;

`-o"<full_path_of_initial_file>"` provides the full path to the file under test or, if it is used, the full path of the project file e.g. `-o"F:\vv70\code"`. If it is not specified, the current directory is used instead;

`-w"<full_path_of_target_dir>"` specifies a directory for the parser's entire output. If this switch is omitted the current directory is used instead. If the directory does not exist it will be created. Useful to specify a target working directory;

`-e"<GNAT_project_switches>"` if a GNAT project file is necessary to compile or browse the source and object files, this switch must be used. It should contain the normal GNAT switches related to project files: `-P` `-X` etc. These will be passed automatically to `gnatmake`, `gnatbind` or `gnatls` by the parser. Compulsory if a project file is present;

`-a"<gnatmake_switches>"` if `gnatmake` needs specific switches for the file under test this switch should be used. Usually not necessary unless compilation problems occur;

`-b"<gnatbind_switches>"` if `gnatbind` needs specific switches for the file under test this

switch should be used. Usually not necessary unless binding problems occur;

-c"<gnatls\_switches>" if gnatls needs specific switches for the file under test this switch should be used. Usually not necessary unless browsing problems occur.

## Output

mika\_ada\_parser generates:

- a parsed file <file\_name\_no\_extension>.po in the new directory : <full\_path\_of\_target\_dir>\<file\_name\_no\_extension>\_mika which is used by the subsequent test data generation phase;
- a subprogram file <file\_name\_no\_extension>.subprograms in the directory <full\_path\_of\_target\_dir> which is used by the GUI front end.

A typical example is

```
mika_ada_parser -M"C:\Program Files\Midoan Software Engineering  
Solutions\Mika\bin" -o"F:\vv70\code" -f"F:\GNAT\2009\bin" -gnat95 -  
w"C:\tmp" -e"-Ptas" alarm
```

which generates the files alarm.po in C:\tmp\alarm\_mika and alarm.subprograms in C:\tmp

## ii. The mika\_ada\_generator the Test Data Generator

The test data generator, mika\_ada\_generator, generates tests according to the given criteria for a specific subprogram in a previously parsed file. It does not check if the file under test needs re-parsing.

The basic syntax is:

```
mika_ada_generator -M"<full_path_of_install_dir>" -  
S<subprogram_name> (-Tbranch|-Tdecision) (-Cignored|-Cnot_ignored)  
[optional_switches] <file_name_no_extension>
```

the -M"<full\_path\_of\_install\_dir>" switch is compulsory and must provide the full path to the bin directory of Mika file e.g. -M"C:\Program Files\Midoan Software Engineering Solutions\Mika\bin";

the -S<subprogram\_name> switch is compulsory and must provide the name of the subprogram for which test data generation is desired e.g. -SCalcEngineCoolingAir the cases are not significant. To handle overloaded subprograms the optional switch -l must also be provided. Operator subprograms may be passed as -S"<the\_operator>" (e.g. -S"\*" or -S"and") or as -Sstring<\_ascii\_code>\* (e.g. -Sstring\_42 or -Sstring\_97\_110\_100 respectively); as mentioned in the GUI front end section, in either cases the <subprogram\_name> will subsequently be known internally in its ASCII form (especially for directory creation);

one of the testing strategy switch (i.e. -Tbranch or -Tdecision) must be provided to indicate

the test data generation testing criterion desired. Refer to the GUI front end section for further details;

one of the context switch (i.e. `-Cignored` or `-Cnot_ignored`) must be provided. If the context is not ignored, the tests generated will be based according to the outcome of the elaboration phase of the package containing the subprogram under test. If the context is ignored, the effects of the elaboration phase will be ignored. Refer to the GUI front end section for further details;

the last argument, `<file_name_no_extension>`, must be the name of the file under test without extension e.g. `:oilandgasmonitoring-gasflowcalibrationcheck`. It can also be `elaboration` in which case tests will be generated to execute the elaboration phase only.

#### Optional Switches:

the `-o"<full_path_of_parsed_file>"` switch provides the full path to the parsed version of the file under test as generated by the `mika_ada_parser` tool e.g. `-o"C:\tmp\alarm_mika"`. If it is not specified the current working directory is used;

`-l<line_number>` is only used to handle overloaded subprograms. It must be used to provide the line number of the first occurrence of the subprogram in the code (could be in its specification or its body) e.g. `-l16`;

`-t` is only used to handle overloaded subprograms. It must be used to provide the line number of the first occurrence of the subprogram in the code (could be in its specification or its body) e.g. `-l16`;

`-t` disables the time stamped directory feature of the output directory. Refer to the GUI front end section for further details.

#### Output

`mika_ada_generator` generates in

`<full_path_of_parsed_file>\<subprogram_name>_year_month_day_hour_minutes_seconds` (or just in `<full_path_of_parsed_file>\<subprogram_name>` if the time stamped directory feature was switched off via the `-t` switch):

- a test results file `<file_name_no_extension>_mika_tests.txt` which contains the automatically generated tests and predicted behaviour of the code;
- a number of Ada files composing the automatically generated tests driver. The main unit of the test driver is contained in the file:  
`mika_<file_name_no_extension>_<subprogram_name>_driver.adb`. It can be compiled using `gnatmake` to generate the test driver executable. Refer to the GUI front end section for further details (especially the requirements for using this feature in terms of test points);

## 8. Examples

The example directory in the installation folder contains a number of small basic examples to experiment with if you do not have access to suitable Ada code.

Feel free to submit your own code examples to Midoan.

## 9. Feedback and Bug Reports

Whether you are entitled to user support under Midoan's maintenance terms or not, Midoan welcomes, and encourages, suggestions for improving Mika. Use the [Feedback](#) button to access Midoan's feature request web page. These may include:

- enlarging OS and/or GNAT version integration;
- adding Ada features not supported in the current subset;
- adding finer user control of test data generation process;
- generating tests for other coverage metrics (e.g. condition, MCDC);
- adding new test output formats for better integration with third party tools.

This is your opportunity to guide the future development of Mika to suit your particular needs.

With Mika's technology being so innovative, and the target programming language so complex, bug reporting is important to us. Midoan will do its best to address identified bugs and issue revisions to all customers as per the appropriate license agreement.