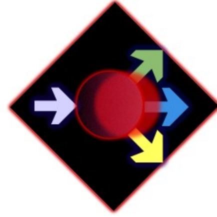


MIDOAN

Mika : Test Data Generation for Ada

Mika

User Manual

Version 1.3

Midoan Software Engineering Solutions Ltd.

May 2020

Copyright ©2020 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 inputs generation tool for code written in a subset of Ada 83, Ada 95 or Ada 2005. Mika uses genetic algorithms to generate inter-subprograms test inputs that will, by construction, exercise, during execution, the maximum possible number of branches, decisions or MC/DCs, in the code under test.

Keywords: Automatic Test Inputs Generation, MC/DC, Unit Testing, Integration Testing, Ada, Spark Ada;

Contents

1 Introduction 3

2 Requirements 4

3 Installation 4

4 Ada Subset 4

5 Limitations 5

5.1 Complexity Limitations . . . 5
5.2 Accuracy Limitations . . . 6
5.3 Miscellaneous Limitations . . . 6

6 The GUI Front End 7

6.1 GUI Area 1: Drive and Folder Choosing . . . 7
6.2 GUI Area 2: Package, Subprogram and Test File Choosing . . . 7
6.3 GUI Area 3: Parsing . . . 8
6.4 GUI Area 4: Test Inputs Generation . . . 8
6.5 GUI Area 5: Progress . . . 10
6.6 GUI Area 6: File View . . . 10

7 The Tools Back End 10

7.1 The `mika_ada_parser` Parser Tool . . . 10
7.1.1 Basic Syntax . . . 11
7.1.2 Optional Switches . . . 11
7.1.3 Outputs . . . 12
7.1.4 Usage . . . 12
7.2 The `mika_ada_generator` Test Inputs Generator . . . 12
7.2.1 Basic Syntax . . . 12
7.2.2 Optional Switches . . . 13
7.2.3 Outputs . . . 14
7.2.4 Usage . . . 14

8 Examples 14

9 Feedback and Bug Reports 15

10 References 15

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 test inputs that will achieve, when executed, the highest possible branch, decision or MC/DC coverage of your code.

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

Mika offers, for the first time, the possibility to automatically generate test inputs 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 test inputs from subprograms that may lead to the execution of 100 000s of line of code. The test inputs generated are 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 [1]: if you have SPARK Ada like code (Mika does not rely on annotations) Mika should be able to automatically generate targeted test inputs 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 inputs 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 test inputs for, pick the subprogram and click the generate test inputs 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 inputs that should, by construction, achieve the highest possible level of coverage of your source code. It replaces a manual process. These test inputs 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 inputs creation.

Mika can generate test inputs that exercise called subprograms as thoroughly as the caller subprogram: the test inputs it generates 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 test inputs it produces are complete and directly executable.

Mika provides, at no extra cost, the complete code behaviour for the test inputs generated: you can validate the behaviour of your code without having to actually execute it if you wish and thus obtain test cases.

Mika provides, with very little additional overheads, an automatically generated

test driver that uses the test inputs automatically generated. This test driver can be compiled and run automatically immediately, or later on for regression testing purposes.

2 Requirements

- Microsoft Windows 10 (Mika is untested for other Operating Systems);
- A working installation of the GNAT Ada compiler (e.g. GNAT Pro, GNAT GPL, MinGW's GNAT).

3 Installation

Download the `Setup_Mika.msi` file from `http://www.midoan.com/download/current/Setup_Mika.msi` and run it. Follow the installation process and first use configuration dialogs. This process is illustrated by a white paper [2] and supporting video [3].

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 (see the release notes [4] for past subset enlargement);
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 inputs generation process will continue. The test inputs generated may still achieve the level of coverage predicted, the code 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);

Static and dynamic code analysis tools often silently ignore some constructs in the code under consideration; we describe below the Ada subset handled by Mika:

1. The subset is based on Ada 2005 as supported by GNAT;
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. Bitwise operators on one dimensional Boolean arrays are not handled;
12. 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. Limitations are common place in static and dynamic code analysis tools, but often hidden; we describe below Mika's limitations.

5.1 Complexity Limitations

The complexity of the control flow graph of the code under test, the size of the data structures handled and the deepness of the coverage desired have a direct impact on Mika's ability to generate test inputs 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 may have a negative impact on Mika's run-time and/or make Mika reach its memory limitations:

1. Large data structures handling (e.g. array objects with more than 1 000 elements).

It is difficult to offer a more precise picture of Mika's complexity limitations. Limitations are being removed as new versions come on stream (e.g. since version 1.2, input dependent dynamic types no longer have a significant impact on the run-time of Mika); see the release notes for past improvements. Mika has been designed to produce results within the reasonable time of the order of minutes: it is not uncommon for the compilation time of the automatically generated test driver to be much larger than the test inputs generation process time itself.

5.2 Accuracy Limitations

The following accuracy limitations are part of Mika:

1. Very large integers are not always handled accurately;
2. Very large floating point numbers are not always handled accurately;
3. Floating point numbers representation is approximate;
4. Boolean operators on arrays of Booleans are not handled accurately;
5. Overloading of Boolean operators with non Boolean returning operators are not handled accurately;
6. `Wide_Characters` are modelled using `Characters`;
7. Intrinsic subprograms are not all handled accurately.

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

5.3 Miscellaneous Limitations

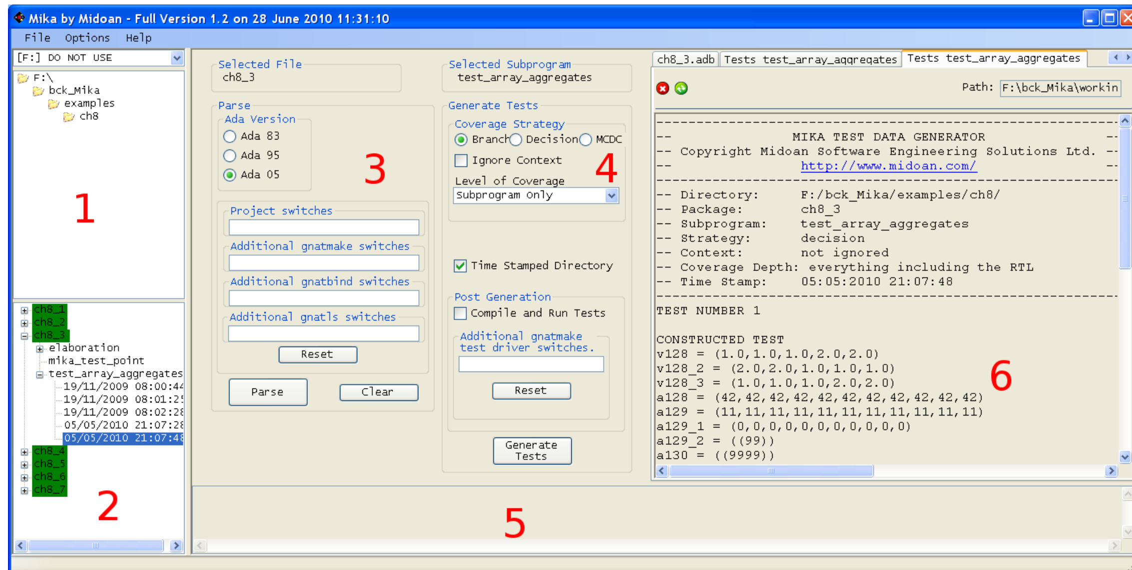
Currently include:

1. Test inputs cannot be generated for subprograms local to another subprogram;
2. The automatically generated test drivers for subprograms within files with krunched names (or that read or assign variables local to packages with krunched file names) do not compile (in other words, although test inputs can be generated for such subprograms, the test driver itself has to be written manually);
3. The automatically generated test drivers for subprograms within nested packages (or that read or assign variables local to nested packages) do not compile (in other words, although test inputs can be generated for such subprograms, the test driver itself has to be written manually). Note that test drivers for subprograms within nested packages are notably difficult to create, even manually and using test points, due to Ada scope restrictions.

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 and/or for use in batch mode).

The GUI is divided into 6 main areas:



6.1 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.

6.2 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 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 inputs generation option). Previous test inputs for a given subprogram can be further expanded and viewed.

By right clicking on a previous test inputs file in this area, you can recompile and re-run the previous, automatically generated, test driver for these test inputs and thus perform regression testing.

Once a package file has been selected its containing file is displayed in area 6;

6.3 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 parser switches of the current package under test. The `Clear` button allows erasing of previous parsing outcomes for the package under test. 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...`.

6.4 GUI Area 4: Test Inputs Generation

Once a subprogram has been chosen, the test inputs generation process can be customised. `Branch`, `Decision` or `MC/DC` coverage can be chosen. Mika implements the masking version of MC/DC [5].

The calling context of the subprogram can also be ignored by ticking the `Ignore Context` option (in other words the initialisations performed during the elaboration process are ignored) or taken into account. If the context is taken into account, the test inputs generated will be after 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. This leads to more realistic test inputs (because real initialisations are used), but may also lead to a lower level of coverage. On the other hand, ignoring the elaboration context allows the generation of test inputs that overwrite the initialisations made at elaboration time: the test inputs may be less realistic but typically will achieve a higher level of coverage.

The `Level of Coverage` desired can be indicated:

- `Subprogram Only` : Mika will only generate test inputs to cover the subprogram under consideration;
- `Local Call Tree` : Mika will generate test inputs to cover the subprogram under consideration and also any called subprograms from the same source file;
- `Entire Call Tree` : Mika will generate test inputs to cover the subprogram under consideration and also any called user written subprograms.

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 test inputs 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 test points 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 test driver will systematically contain the context: `with System; use System;`. In effect, 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 test inputs according to the user settings. Progress is shown in the area 5. Actual test inputs and code behaviour are displayed in area 6.

6.5 GUI Area 5: Progress

Warnings and errors are displayed in this area.

6.6 GUI Area 6: File View

This area contains a simple file viewer to view (but not edit, to minimise the risk of corrupting production code) source code and generated test inputs. The generated test inputs and code behaviour are contained in a file named `mika_<subprogram_name>_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) or to use Mika in batch mode.

There are two tools provided:

- `mika_ada_parser` the parser tool;
- and `mika_ada_generator`, the test inputs generator tool.

7.1 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 inputs 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, the parser tool 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.

7.1.1 Basic Syntax

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

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

where 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 e.g. `-M"C:\Program Files\Midoan Software Engineering Solutions\Mika\bin"`;

one of the Ada version switch (i.e. `-gnat83`, `-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`.

7.1.2 Optional Switches

The optional switches of the parser tool are:

`-f"<full_path_to_gnat_bin>"` provides the full path to bin directory of the GNAT version you wish 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. This switch is useful in order 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. This switch is 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. This switch is 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. This switch is 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. This switch is usually not necessary unless browsing problems occur.

7.1.3 Outputs

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 inputs 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.

7.1.4 Usage

A typical usage example of the parser tool 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

7.2 The mika_ada_generator Test Inputs Generator

The test inputs generator, mika_ada_generator, generates test inputs 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.

7.2.1 Basic Syntax

The basic syntax of the test inputs generator is:

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

where the order of the switches is not significant.

- 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 inputs 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); 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`, `-Tdecision` or `-Tmcdc`) must be provided to indicate the testing criterion desired;
- one of the context switch (i.e. `-Cignored` or `-Cnot_ignored`) must be provided. If the context is not ignored, the test inputs 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 just `elaboration` in which case test inputs will be generated to execute the elaboration phase only.

7.2.2 Optional Switches

The optional switches of the test inputs generator are:

- 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. -l96;

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

-u<S|F|A> indicates the level of coverage desired: S : Subprogram Only, F : Local Call Tree, A : Entire Call Tree. E.g. -uF. If it is not specified -uS is the default. Refer to the GUI front-end section for further details.

7.2.3 Outputs

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

- a test inputs result file mika_<subprogram_name>_tests.txt which contains the automatically generated test inputs and code behaviour;
- a number of Ada files composing the automatically generated tests driver. The main unit of the test driver is contained in the file:

  mika_<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);

7.2.4 Usage

A typical usage example of the test inputs generator is:

```
mika_ada_generator
  -M"C:\Program Files\Midoan Software Engineering Solutions\Mika\bin"
  -Stomorrow -Tmcdc -Cignored -uF -t array_date
```

which generates the files mika_tomorrow_tests.txt and
mika_tomorrow_driver.adb in the directory
<full_path_of_parsed_file>\tomorrow_8

8 Examples

The example directory in Mika's 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 from the GUI . 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 inputs generation process;
- generating test inputs for other coverage metrics (e.g. condition coverage);
- 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.

Midoan will do its best to address identified bugs and issue revisions to all customers as per the appropriate license agreement.

10 References

[1] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, 2003.

[2] Midoan Software Engineering Ltd. Basic automatic generation of white box test inputs from ada source code using Mika. Technical report, 2010.

[3] Midoan Software Engineering Ltd. Mika presentation page. `http://www.midoan.com/mika.html/`.

[4] Midoan Software Engineering Ltd. Mika release notes. Technical report, 2010.

[5] K.J. Hayhurst, D.S. Veerhusen, J.J. Chilenski, and L.K. Rierson. A practical tutorial decision coverage on modified condition/decision coverage. Technical report, NASA, 2001.