BSSE System and Software Engineering

Software Diversity by Automation

This paper was presented during

DASIA'05: Data Systems in Aerospace

May 30 - June 2, 2005, Grand Sheraton Hotel, Edinburgh, Scotland

organised by Eurospace, Paris

ESA Document SP-602

BSSE System and Software Engineering

Auf dem Ruhbuehl 181
88090 Immenstaad, Germany

Phone +49/7545/91.12.58
Mobile: +49/171/80.20.659
Fax +49/7545/91.12.40
e-mail: Rainer.Gerlich@bsse.biz
Ralf.Gerlich@bsse.biz
Thomas.Boll@bsse.biz

URL: http://www.bsse.biz

Software Diversity by Automation

Rainer Gerlich Ralf Gerlich Thomas Boll	Klaus Ludwig	Philippe Chevalley	Neil Langmead
BSSE System and Software Engineering	ESA/ESTEC ATV SW Section MSM-MAS	ESA/ESTEC TEC-SWE	IPL Information Processing Limited
Auf dem Ruhbuehl 181 88090 Immenstaad Germany	139 route de Verneuil 78132 Les Mureaux France	Keplerlaan 1 - Postbus 299 2200 AG Noordwijk ZH The Netherlands	Eveleigh House, Grove Street Bath, BA1 5LR, UK
Phone +49/7545/91.12.58 Mobile +49/171/80.20.659 Fax +49/7545/91.12.40 e-mail Rainer.Gerlich@bsse.biz Ralf.Gerlich@bsse.biz Thomas.Boll@bsse.biz URL http://www.bsse.biz	Phone +33/1/5369-8040 Fax +33/1/5369-8079 e-mail klaus.ludwig@esa.int	Phone +31/71565-6539 Fax +31/71565-5420 e-mail philippe.chevalley @esa.int	Phone +44/1225/475000 Fax +44/1225/444400 e-mail Neil.Langmead@ipl.com URL http://www.ipl.com

Final version dated 18.06.2005
DASIA'2005, Edinburgh, UK, May 2005

Abstract: Code diversity is applied to critical software for identification of bugs which may hide during tests of a single implementation. The principal idea of diversification is that (1) independent implementations of the same requirements should yield the same results, (2) consequently, any deviation in the results indicates a bug in one of the implementations. Usually, all implementations are performed manually by independent teams requiring a lot of effort. In our approach we expand code diversity to platform and testing diversity. We consider two implementations and replace the second team by an automaton to save effort and time. We apply this approach to the flight application software (FAS) of ATV which is written in Ada83.

1 INTRODUCTION

Code diversity for the FAS would be very expensive because of its huge size. Therefore we sought for possibilities which allow performing similar checks as for code diversity but at little expense. At BSSE we have experience with automatic analysis and generation of code. Therefore we got the idea to achieve diversity by use of automata, and to generalise the concept of diversity.

In case of n-version programming the principal idea is that the same inputs should yield the same outputs whatever implementation produces the results. However, this principle is not necessarily limited to comparison of application software developed from scratch. To evaluate the validity of results we may compare messages from any independent, but equivalent programs. We are now extending the scope of diversification and consider the following applications areas:

- code diversity, for new implementations (also addressing the "classical" n-version programming) by generating independent, but equivalent versions of code,
- platform diversity, comparing results obtained from independent platforms,
- testing diversity, comparing results of independent test stimulation.

Information is needed for generation of another but equivalent version, porting of an existing version to another platform, or generation of a set of test cases, which allows derivation of the desired result. Therefore we have to execute two principal steps: extraction of existing information, and transformation of this information into the final, additional product. Usually, both steps are executed manually. To keep costs low for implementation of another version, we are automating the analysis and the transformation step. We apply this concept to the FAS of ATV in order to demonstrate its feasibility and to support the project by additional verification results. An intermediate version of the FAS was used, not fully validated but enough for the experimentation.

For execution of this work either we need to establish new tools, e.g. for code analysis and porting of the software, or to extend existing tools, e.g. for code generation and testing. These tools will be commercially available soon to support any Ada application. The following types of diversification are supported:

1. Platform diversity

   The code is ported to a different compiler (running on another processor type and operating system), and compiler messages are analysed.

   The existing source code is adapted automatically to make it compilable on the second platform, without changing its logic and functionality. In case of the FAS nearly 1500 (automated) changes were needed spread over about 1 million lines.

2. Code diversity

   The behavioural specification is extracted from the existing code and transformed into a notation which allows to check its completeness and consistency, and to build an independent real-time skeleton into which the remaining (sequential) Ada code can be plugged-in.

   In our case – the extraction of information on FSMs – the analysis tool needs to know where to find the FSMs in the Ada code, i.e. this part is application-specific.

3. Diversity of test case generation

   Two different approaches on test case generation are applied:

   - testing of Ada subprograms ("unit testing")

     Test cases are automatically generated and executed which are derived from the specification of the Ada subprograms. Results are monitored automatically.

     Due to automation a huge number of test cases can be generated and evaluated. Also, faults can be injected.

   - behavioural testing

     Test cases are automatically derived from the behavioural specification and the FSMs of the software system are stimulated. Results are monitored automatically.

     The full set of expected (incoming) messages can be generated and evaluated.

     Behavioural testing requires a complete and consistent set of FSMs, but not necessarily a full implementation of the algorithms. Only the impact on states by the algorithms needs to be considered for test case generation to achieve a representative operational scenario.

   Faults, i.e. out-of-range data and unexpected messages, can be injected, too, to exploit the robustness of the software ("robustness testing").

2 PLATFORM DIVERSITY

During previous tests on different platforms, i.e. different types of processors, different operating systems and different compilers, we have observed that each platform acts as indicator for certain types of bugs. Bugs may be detected immediately and at little effort on one platform, while it is impossible to identify them on another one, or only at high test effort. Hence, diversity of platforms allows detecting and removing bugs in an efficient manner. Any additional differing combination of the components "processor", "operating system" and "compiler" may be helpful to identify a certain type of bug.

For example, the GNU C and C++ compilers do not inform on non-initialised variables, while the Visual C++ compiler (VC++) does not warn in case of unused variables. By moving between Linux/GNU and MS-Windows™ / VC++ platforms we were able to easily identify such deficiencies of the source code. Similarly, we have observed that bugs related to pointers will be identified when moving between PC/Linux/GNU, PC/VxWorks/GNU, Sparc/Sun-Solaris™/GNU and Sparc-SPLC/VxWorks/GNU (SPLC = ESA Standard PayLoad Computer).

Therefore we decided to port the Ada code of ATV FAS from an FTC/Aonix (Fault Tolerant Computer system, three lanes, voting) to a PC/GNAT platform and to analyse the GNAT messages – if any ([Aonix], [GNAT]). We established a program, called "AutoPort", which transforms statements incompatible with GNAT from Aonix to GNAT notation. The statements to be adapted were identified by the GNAT compiler messages and associated actions were defined.

This is a non-exhaustive list of messages which were issued by the GNAT compiler:

1. missing packages (error)

   Specific packages provided by the Aonix compiler have to be provided for GNAT.

2. inconsistent filenames (error)

   Package name and filename must be identical for GNAT or a pragma must be provided mapping package and file name.

   This is a constraint imposed by GNAT.

3. syntax of "PRAGMA IMPORT"

   GNAT uses another syntax than Aonix for the import pragma. A variety of IMPORT statements had to be covered.

4. priority, illegal range (error)

   GNAT supports the range 0 .. 31, while Aonix allows 0 .. 255. The priorities were mapped onto the GNAT range by a linear transformation. Mapping of priorities, which are different in the Aonix range, onto the same GNAT priority is checked and reported. In case of FAS we did not run into such a conflict.

5. unchecked conversion, types have different size (error)

   There is an ambiguity in Ada standard regarding the data representation. For an "unsigned integer" with range 0 .. 2^{15} related to a "first named subtype" the Aonix compiler returns 16 bits as size while GNAT returns 15 bits. When dividing this result by 8 to derive the number of bytes, different results are obtained. If an array is created by taking this result as range, the length of the source type and its byte-array representation are different. Consequently, an error occurs, when both types are passed to "unchecked_conversion".

   The problem was solved by replacing the (standard) SIZE-attribute by OBJECT_SIZE only supported by GNAT, which returns the expected size of 16. This is the object's size in memory, while 15 is the real number of occupied bits.

6. unused variable

   A local variable (stack) is not used within a subprogram (warning).

7. missing case value "xxx"

   A case ("when =>") is missing in a list of "when" for an enumeration type (error).

   Fortunately, the ATV software standards forbid to use the "others" clause. Therefore such an error can be detected.

Message types 1 – 4 are related to compiler-specific support or implementation of some Ada features. While (1) and (2) are really a matter of the compiler, (3) and (4) are a consequence of compiler-dependent solutions allowed by the Ada standard. If constraints like (2) are known in advance, file names and package names can be harmonised right from the beginning.

Message type (5) is a matter of an ambiguity of the Ada specification (Ada83 and Ada95). For more details see e.g. ARM95 [ARM95], sect. 13.3, clauses 39 ff, especially clause 55. As the used types are "first-named subtypes" no pre-defined size can be assigned by a representation clause. Consequently, this is a counter-example on the hypothesis: "Any (pure) Ada program will produce the same results on any Ada platform".

Message types (6) and (7) are real software engineering issues. Type (6) is just a warning, but indicates where storage can be saved. It becomes more critical e.g. when another message occurs saying that a loop variable of the same name is used. This may indicate a conflict due to limited scope of variables.

Type (7) indicates a bug, which may require high effort to identify by testing or by code analysis within a context of some hundreds KLOC.

AutoPort does port the code by iterations of adaptation and compilation (Fig. 2-1), because the source of some error messages must be removed, before another type of error messages or warnings will be reported by the compiler. For each step we analyse the compiler messages and initiate the corresponding actions on code adaptation. Then we start the next compilation step. This is repeated until the compiler messages do not change any more or no message occurs at all. Each modification is tracked and reported. The observed messages are auto-documented for each step, in detail and as summary report, together with previous porting results for comparison.

We observed 8 to 9 iterations (depending on the actual version of the ATV FAS) executed within about 20 minutes on a PC/laptop 1.6GHz, which results in about 2 minutes to compile and adapt an amount of about 900 KLOC (in total) of Ada code spread over about 450 files. The RTF-document is immediately available at the end of porting, i.e. after about 20 minutes. In addition, graphical information on package dependencies may be attached, i.e. "package x uses packages ..." and "package x is used by packages ...".

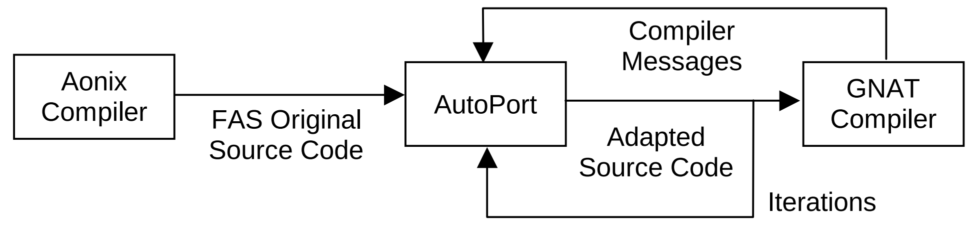

Fig. 2-1: Porting and adaptation of source code

AutoPort is available for any other Ada software compilable on an Aonix platform. It may be extended to any other source and target platform, if requested for.

3 CODE DIVERSITY

In case of manual development code diversity is achieved by independent teams implementing the same requirements. If we can feed a formal specification into different code generators, we will get code diversity, too, at low costs and the code and results are available within a short time.

In principle, we will apply this approach, but we have to modify it slightly because a formal specification is not available for FAS, while the code already exists. Therefore we decided to derive the formal specification from the existing code. This way we can verify the extracted information and generate an independent implementation, which can be executed on another platform. However, we do not check whether the derived specification really complies with the (informal) original specification of FAS.

Due to limitation of the budget we decided to concentrate on check of behaviour, because this seemed to be the most urgent issue. The FAS of ATV uses Finite State Machines (FSM) on top-level to control the logic flow of processing. Therefore an independent implementation of behaviour and the real-time infrastructure can immediately be generated by the ISG tool [ISG], which converts (extended) FSMs into executable code ("RT Environment", see Fig. 3-1). The remaining part of the FAS, which is purely sequential software, shall be plugged into the skeleton provided by ISG, using an ISG facility for automated integration of function calls ("interfaces"). In addition, DARTT supports instrumentation of the ported code for recording of coverage. This will allow to compare the results of real-time execution and message exchange, and to evaluate execution of the ported (sequential) source code on another platform.

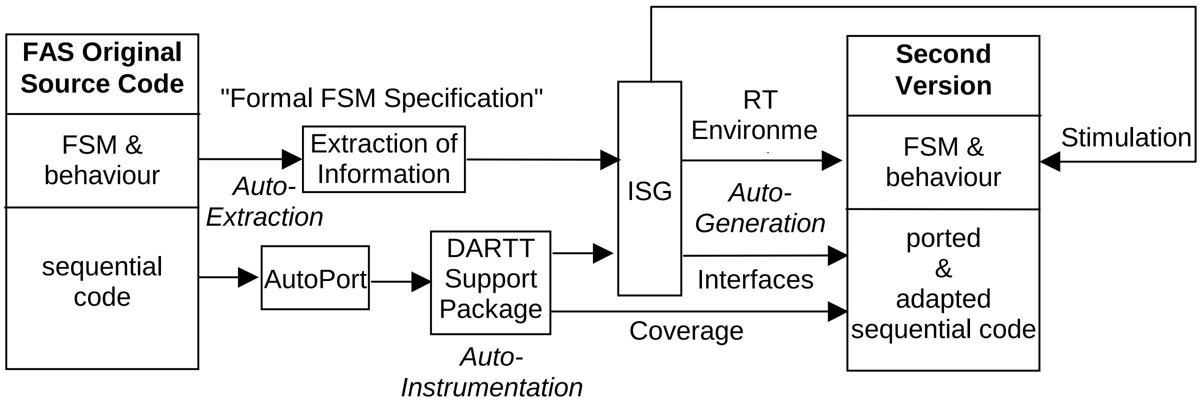

Fig. 3-1: Generation of a second version for GNAT platform

ISG expects a specification of behaviour (expressed by extended FSMs), distribution of processes across a network, performance and communication, checks this information for correctness and converts it into executable code, which is complemented by a test and validation environment. The infrastructure of a real-time system consisting of e.g. 40 processes can be correctly generated on a PC within about 10 minutes without manual intervention. At the end of a run a report on the properties of the system is available, consisting of text files, graphics (timing, message exchange) and an RTF-document. We hope that we will have performance results available of this exercise to be presented at the conference.

The FAS processes are scheduled periodically at 10 Hz, 1 Hz and 0.1 Hz. 100 time-slots are provided at 10 Hz, and each process gets 1, 10 or 100 time slots depending on the frequency it has to be scheduled. As execution of the algorithms may take more than one time slot, they are broken down into a number of pieces which can be executed one after the other in consecutive time-slots of the associated process ("interrupted execution").

In the environment generated by ISG we may investigate two different scheduling approaches: (a) the original one as used by ATV (interrupted execution of algorithms), and (b) non-interrupted execution, but prioritizing the processes under the OS, i.e. applying pre-emptive scheduling. This will allow analysing how well deadlines can be met in case of usual scheduling. This scheduling analysis is not part of the verification, but is just of interest for comparing time-slot and pre-emptive scheduling.

A bridge is needed which extracts the information from the existing Ada code and generates inputs for ISG. We will use ASIS (Ada Semantic Interface Specification) to get larger portions of information, and apply own analysers to get the detailed information. We recognised that ASIS/GNAT is rather slow and it is rather difficult to parse the information with Ada programs. As we have already available a lot of C programs for parsing of information, we decided to perform the detailed analysis in C.

ASIS is a tool provided by Ada compiler environments to support code analysis.

"Code diversity" is an on-going activity.

4 AUTOMATIC TEST CASE GENERATION

Test cases can be generated automatically from a formal specification, too. This allows complementing the tests planned and executed by the ATV project. Two different types of test activities are considered:

- subprogram testing

  In this case the formal specification is the prototype of an Ada subprogram.

  Information on subprograms is extracted from Ada code of the FAS. For each subprogram a test environment is built. For each input parameter values are generated, either randomly or incrementally.

  "Random selection" of test cases means: values are selected randomly from the given valid or invalid range of a parameter yielding a uniform distribution.

  "Incremental selection" of test cases means: each range is divided into a given number of intervals, each end point of an interval is selected as test case, and such test points are selected in an ordered manner starting by the FIRST-value of a range. In principal, we map the point m/n of [0,1], where n is the number of intervals and 0≤ m ≤ n, onto [FIRST,LAST] of a type range. In case of a structure or an array we apply m/n to each component.

The required functions for initialisation, test case generation and test evaluation are generated automatically for each user-provided type. The DARTT tool [DARTT] is used for this activity.

- behavioural testing

  In this case the formal specification is represented by the FSMs implemented in the FAS Ada code, expressed by data tables.

  The information on FSMs is evaluated and test cases are derived to stimulate the FSMs, also covering fault injection, i.e. injection of erroneous commands.

  Test coverage of FSMs, performance properties and other characteristics are evaluated. MSCs (Message Sequence Charts) [MSC] and timing diagrams (timer events and exchanged messages recorded over time) are provided. This testing is based on the capabilities of ISG and will be executed in the real-time environment as described in the previous chapter.

4.1 Subprogram Testing

4.1.1The Principal Approach

DARTT automatically generates test cases for parameters to stimulate Ada subprograms. It is now available for GNAT and has been extended in its functionality. Profiles of execution time, test coverage on block level (begin-end, if-then-elsif-else, loop, case-when, exception-when), occurrence of unhandled exceptions and 1st derivative of output vs. each input parameter are recorded and evaluated. Graphical figures on input and output data are provided. For each function the results are reported in an automatically generated RTF-document (Fig. 4-1).

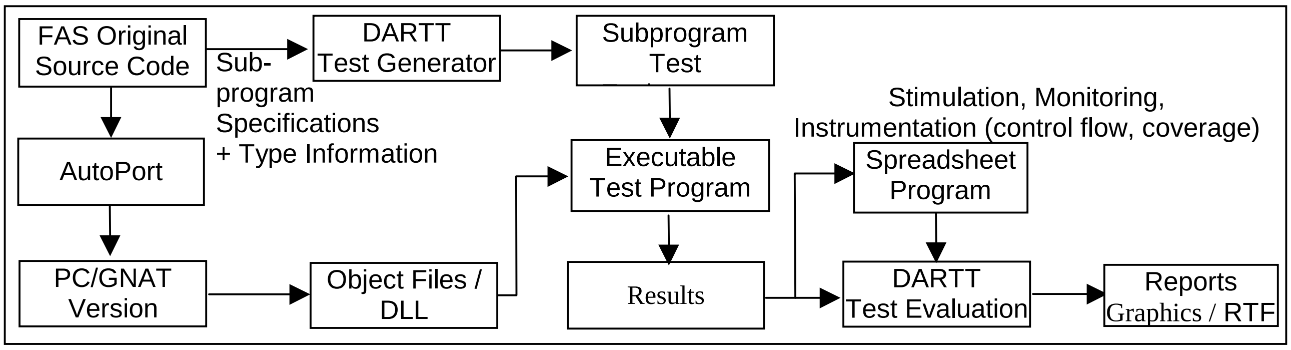

Fig. 4-1: Automatic Test Case

It analyses the types and the subprogram specifications, builds a test environment for each subprogram, executes and evaluates the results. Test data are generated over the full range of each input parameter, and also outside of this range in case of fault injection. Amongst others, test goals are to check if (1) unhandled exceptions occur within the valid data range and (2) erroneous input data are tolerated.

Usually, we expect that the same exceptions would be raised on all platforms, e.g. constraint errors due to an invalid range, program errors may be raised due to missing elaboration and so on. This is the basic set of "deterministic" exceptions, which always will occur on all platforms due to execution of algorithms or deterministic timing conditions. However, there are also exceptions which will be raised by non-identified bugs. Not all bugs can be identified at compile time, e.g. by AutoPort when moving to another platform. Hence, some bugs will "survive" the checking capabilities of the "platform diversity"-approach (see chapter 2 above), and will occur at run-time.

Such a typical exception is e.g. a STORAGE-ERROR, raised when a value shall be assigned to a variable which has been declared as "constant". We do not understand why such an assignment is not / cannot be already identified at compile time, yet we observed it occurring at runtime. It may happen that one compiler detects the bug, the other does not. To catch such bugs, it is essential to execute the ported code, not just to compile it on another platform.

Such an exception may be raised when a (dummy) variable is mapped onto another variable (to which storage is allocated) by a number of "renames". Then it cannot be seen immediately by an engineer that the variable is protected, because he has to look into a number of packages. So it may happen – possibly due to a compiler bug – that on the highest level the (dummy) variable is NOT protected by its declaration, though the real variable IS protected. Then assignment of a value on top-level is not rejected by the compiler.

Such a high number of "renames", i.e. in fact non-readable code, may be created by code generation tools, which support package hierarchies. In this case subprograms or data may be passed top-down through the hierarchy to the physical item. If conflicts like the one described above for a "constant"-declaration are not identified by the generator and the used compiler, a crash may/will occur at run-time when the statement is executed. This raises the need for exhaustive testing as independent means for bug identification. It may happen that such illegal code is tolerated by a compiler, and the contents of a variable can be changed in RAM, though it should be protected. Correct code cannot be expected in this case.

Taking an existing software system as black-box and plugging its components into a test environment requires some additional measures, especially in case of an embedded and/or real-time system. For such a system the environment has to be emulated and proper data have to be provided. A system may expect an initialisation command from a channel and will hang, e.g. during elaboration, if it does not receive it. Then the main test procedure will never get control. Similarly, tasks may be created during elaboration and will create undesired CPU load, while a certain subprogram is tested.

Our goal is to solve such problems automatically without the need for any other information than what is available in the source code, by type definitions and subprogram specifications. We have now implemented an approach which allows us to take any such Ada code and to prepare it by AutoPort for subprogram testing, avoiding blocking as described above.

Also, other problems have to be solved when a large amount of tests shall be executed. The FAS includes thousands of subprograms, i.e. a high number of test environments has to be built, and for each such environment again a high number of test cases has to be generated. Due to the large number of symbols in the object files of FAS it takes about 1 minute on the average on a PC-1.6 GHz mobile to prepare, compile and link the files. Consequently, it takes several days just to build all environments. Due to this large amount of time required even for automatic preparation of tests, tuning of performance is a challenging task, which we are already tackling.

For test execution we have measured a rate of 10 Mtests/h for the example sin-function including recording of test data.

A large amount of test cases can be generated this way, e.g. to analyse the robustness of a subprogram. However, it is difficult to assess on the correctness of the results, because usually no oracle is available which predicts the expected results. But there are cases for which the expected output may be known:

- a user can manually provide results for a number of inputs,
- reliable results are available by another application, e.g. by simulation or by a previous version,
- an inverse function exists, by which the outputs can be transformed back into the input domain.

In all these cases the test environment automatically built by DARTT will help to prepare testing, and to save costs and time. If a user knows the results, (s)he can provide this information through an interface to DARTT, which will compare observed and expected results for the given number of test cases. In case an inverse function exists, DARTT will pass the output values to it and compare the output of the inverse function with the provided inputs. The mapping between the outputs of the function-under-test and the inputs of its inverse function needs to be provided by a user – in case more than one parameter has to be passed.

Finally, if none of above options exists, DARTT provides tables of input and output data which can be evaluated manually (of course only for a limited number of test cases, depending on the budget limits).

It is important that DARTT provides compressed and comprehensive test reports. It does not make sense that a user needs to evaluate all the produced data, which may amount to millions of data sets. Files may have a size of several MB, resulting in some GB for thousands of tests. Therefore DARTT provides summary reports, which only include anomalies like unhandled exceptions and discrepancies between observed and expected results. Then based on this information more details can be analysed when needed.

For a recent test of about 5200 Ada subprograms - running about 3½ days - about 15 GB disk space was occupied, the document amounts to about 660 pages, automatically generated. This raw information has to be filtered and presented for several criteria like "exceptions" or "coverage" in order to ease understanding its contents. The related data sets – input and output data vectors – can be accessed as supplementary information from the document directly. The large amount of raw information has to be compressed for later use. Manual comparison of results amounting to more than 600 pages would be very time consuming. Therefore indicating the difference to a previous version will make life easier for those who have to evaluate the results and to derive conclusions.

The DARTT tool shall be integrated with AdaTest from IPL to complement the functionality of both tools.

4.1.2 Results

4.1.2.1 Identification of Potential Risks

DARTT identified a number of locations in code where deeper consideration is needed. The analysis of the code for a first set of raised exceptions showed that no risk is related to them. We do hope that complete analysis of all identified locations will yield the same result. In principle, this result is not surprising, but gives a good or better feeling on the automatically code. It is not surprising because all the usual tests procedures have been applied before and in addition Independent Software Verification and Validation (ISVV).

The raised exceptions occurred for code which could not be verified statically based on the syntax and semantics of Ada. E.g. the following piece of code is potentially unsafe.

```
i,k :natural;
k:=i+1;
```

Why is it unsafe? To get an answer to this question consider the case: i:=natural'last. It is obvious that a "constraint exception" will occur. DARTT will enforce occurrence of such an exception, because it will feed in data over the full range of a variable including "type'first" and "type'last" (this is also true for all elements of arrays and records).

The exception raised by DARTT tells us that we need to confirm that "i" never can take the value "natural'last", or we need to provide error handling by an own check (as shown below), or by an exception handler catching the exception.

```
i,k :natural;
if i=natural'last then
  <error handling or some other actions>
else
  k:=i+1;
end if;
```

A reader may argue that "usually" the variable "i" never will take such a high value. The message from DARTT in this case is: yes, could be, but do prove it. In particular, this is of relevance, because "i" is a subprogram parameter. Therefore only the caller knows whether the actual value is correct or not – as far as the subprogram does not check on valid data itself. Consequently, verification, that "i" can never take a wrong value, implies deeper investigation of all calls – and increase of effort.

The other point is: if "i" never can take such a high value, why a more specific type is not defined expressing the limited range? The answer is: to avoid exceptions when calculating expressions which are close to the boundaries of a range. However, when using types with large ranges, the static checking capabilities of Ada compilers are lost for verification of the code. In such a case a compiler cannot detect at compile-time that "i" never exceeds the allowed range. The compile-time checks need to be replaced by run-time checks and testing, which is much more expensive. So a good solution would be in the case discussed:

```
myUpLim : const natural :=10;

subtype myTypeIn   is  natural range 0 .. myUpLim;
subtype myTypeOut is natural range myTypeIn'first .. (myTypeIn'last + 1);

i  : myTypeIn;
k : myTypeOut;

k:=i+1;
```

Then the compiler could already recognise an out-of-range condition when assigning an invalid constant or data of another type. Of course, this requires more coding effort, but saves a lot of later test and analysis effort. In general, such problems do occur for any expression, and makes proper handling of types complex. A reasonable solution would be to cast data from a user-defined type to the type covering the largest range needed for the calculation before executing the expression, and to cast it back to a user-defined type. While the forward-conversion is uncritical, the backward-conversion is not. Due to a bug the result may not be compliant with the type range, and an exception may be raised at run-time. This indicates the second principal case where code is potentially unsafe: a type cast, which will be discussed later in detail.

```
type TyMyEnum is (enum1,enum2, ..., enumN);

procedure myProc(para1 : in natural) is
  procEnum : TyMyEnum;
begin

  myEnum:=TyMyEnum‘val(para1);

end myProc;
```

Also, DARTT identifies whenever the specification ("prototype") and the body of a subprogram are not compliant. Consider the case shown in the box. For whatever reason (e.g. specific types may not fit with an overall architecture into which the subprogram is embedded) the type as used in the body is not used in the specification. When DARTT stimulates the subprogram over the given range of the parameters – "natural" in this case, an exception will be raised because the range of the enumeration will not allow natural'last, for sure. Again, such an exception is an indication or a reminder that a proof on the correctness of the parameter cannot be done by the compiler, and additonal checks at run-time are needed to become aware of an out-of-range condition. The Ada checking mechanisms should be activiated or own checks be inserted.

In addition to the already described potential problems, DARTT can identify the following potentially unsafe code (this is a non-exhaustive list):

1 type casting

In this case the range of a subprogram parameter does not comply with what is expected inside the subprogram. To be more general, we found an unchecked conversion. There are two principal sub-categories:

1.a conversion based on the val-attribute ('val)

used to convert to an enumeration type, e.g. from natural. In this case it is not clear whether the bounds of the enumeration type will be exceeded or not when calling the subprogram.

1.b type conversion based on pointers,

e.g. using the address-attribute ('address)

used to map a data stream onto another type, e.g. to map a character stream (buffer) onto a specific structure. As DARTT only sees the wrong external type, e.g. a string type, the data cannot be initialised properly. If the contents of the expected data structure is not checked by the subprogram, an exception will be raised, of course.

2 memory alignment

In this case the address of the data is calculated dynamically at run-time, e.g. to move a "data window" over a data stream. Two types of faults may occur:

2.a misalignment

A variable (access type) may not be allocated at the proper address which should be a multiple of its size, e.g. an odd address may be assigned to a two-byte type. The related type of the parameter does not exclude values which cause misalignments.

Misalignment of data is not necessarily a functional defect on all platforms. The Intel 80x86 processors, e.g., are able to read and write misaligned data, but misalignment has a negative impact on performance, which is why these processors allow to optionally detect such misalignment using a processor exception. On the other hand the Sparc architecture requires data to be properly aligned and is unable to read or write misaligned data.

2.b out-of-bound

The calculated address may move the data window out of the allowed range. Similarly, the related type of the parameter does not exclude values which cause an out-of-bound condition.

3 potential out-of-range

e.g. during or after calculation of an expression like explained above for k:=i+1

4 storage error

A value is assigned to data which are declared as constant. In certain cases an Ada compiler cannot identify such an inadvertent assignment at compile-time. Then such a bug can only be detected bug during test execution. It may not necessarily cause a problem. E.g. when the data are stored in RAM, no fault will occur, in fact. However, it is a fault when being stored in ROM. Such an assignment may even be erroneous from a logical point of view when being stored in RAM, e.g. when constants are destroyed by undesired assignment of data.

Other types of exceptions may be raised due to a principal problem of dedicated subprogram testing. When calling a single subprogram, execution of this subprogram may require initialisation of data prior to its execution. Similarly, test of a stack-pop-operation requires some data on the stack, and usually calls of other subprograms which condition the stack. Such problems can be solved by definition of call sequences of several subprograms, which are encapsulated in another subprogram, e.g. by using ATTOL, AdaTest or own Ada code.

Dedicated subprograms containing such call sequences – so-called mockup subprograms – may also take the place of the actual target subprogram in test. Instead of generating test input for the target subprogram, test cases for the mockup subprograms are produced. These mockup subprograms therefore represent sets of specific testcases for which again tests can be produced automatically – in contrast to similar constructs in ATTOL, AdaTest, Aunit or similar, which only represent manually selected test cases.

In addition, Ada provides the capability of elaboration, which may be useful to call the initialisation subprograms in the elaboration part. This way proper initialisation of data is always guaranteed.

4.1.2.2Performance of Test Case Generation

About 85 hours were needed to build 5204 test environments for all the subprograms included in FAS and to execute 3000 test cases for each subprogram (or some more in certain cases). On the average about 1 minute was needed to build a test environment on a PC-laptop 1.6GHz. Most of the time was spent for compilation and linking, which took between about 20 seconds and 15 minutes, while test execution required between 1 .. 20 seconds, only. Compilation and linking takes a rather long time because a high number of symbols have to be managed.

Of course, we are continuously looking for measures by which the overall execution can be reduced. When the average duration of a subprogram test is changed by 10 seconds only, this impacts the overall test duration by about 15 hours in case of FAS. However, we have also to consider this strong dependency of overall duration on a few seconds per tests when improving and extending DARTT. It would be highly desirable to save a number of days, not having to wait 3½ days, but e.g. 1 day instead. So we take this issue as a challenge. Nevertheless, this issue also indicates which effort can be saved by automation. When dozens or hundreds of man-years are needed to build such test environments manually, it does not make sense, at all, to ask for a reduction of a few days.

The number of test cases per subprogram was limited to 3000 due to constraints on disk storage. 3000 test cases leads to more than 15 million test records which need to be stored together with other information. In extreme cases up to 64 million test cases were generated. Storing of test vectors is an option for later manual inspection and graphical display of test data. When disabling this option much more test cases can be executed without being constrained by available disk space.

As a lot of information may be produced on potentially unsafe locations in code, a comprehensive presentation of results is a "must". A number of exceptions may be raised for the same reason, occurring in a number of files. Grouping such information according to different criteria helps to analyse this information and to draw conclusions, not being forced to consider every location when deriving principle decisions on how to handle such observations. Therefore DARTT provides documentation (in rtf-format) in detail and as summary information on

- files / packages, for which exceptions or other anomalies occurred, or for which the achieved coverage is < 100%
- location of exceptions sorted by the criteria exception type, exception message, exception category and filename / package (the categories may be user-defined),
- coverage of functional blocks and execution of inspection points,
- statistical information on the observations.

(Hyper)Links correlate pieces of internal information, and allow easy and fast access of external files like source files or files including more detailed information about the observations.

Occurrence of exceptions leads to reduced coverage. To increase coverage in presence of exceptions, DARTT is currently extended to replace automatically critical statements raising exceptions by such ones which provide valid data instead, so that the test can be continued. Such test sessions can be executed in iterations. DARTT identifies which test runs raised exceptions, and builds a new set for another test execution cycle. This procedure shall be repeated until no exception does occur at all. At the end the maximum test coverage is obtained which can be achieved by stimulation of subprograms, i.e. black-box tests. This extension also covers a.m. cases where an exception is raised e.g. due to missing initialisation or unchecked conversion, i.e. cases which do not well fit with the concept of subprogram stimulation based on a dedicated call.

4.1.2.3Conclusions on Automatic Subprogram Testing

The presented test approach allows efficient identification of potentially unsafe code at a measured rate of about 1.5 LOC/s (about 5000x60s=300,000 seconds for about 430,000 LOC) for a rather large application including building of about 5200 test environments, and compilation, linking, test execution and analysis for each environment. On the average about 1 minute was required to build and run each of the 5200 tests. Test vectors may be generated for later manual evaluation, optionally.

Efficient and automatic organisation of the derived information is needed, to keep the remaining manual analysis effort low. The efficiency of automatic testing can be increased by proper organisation of the source code. Reduction of test duration by a few days does not matter for usual manual testing at all – because it can be neglected compared to the total huge effort – but it becomes a tuning issue, when the overall test duration is reduced to a few days by automation.

4.2 Behavioural Testing

The test goal is to investigate the system behaviour when being stimulated by valid and invalid external commands, to record the actions, exchanged messages, timing profiles, and to identify anomalies – if any.

The system-under-test consists of the real-time infrastructure generated by ISG and the ported sequential code (see chapter 3).

This activity requires ported code which has been subject of heavy subprogram testing in order to avoid unexpected problems at run-time which are difficult to identify in an integrated environment.

By injecting arbitrary sequences of commands, automatically generated, we will try to stimulate the system with "unusual" sequences, i.e. sequences which are not considered as reasonable and therefore not generated manually, and evaluate the system reaction. Of course, we do not know yet, what an "unusual" command is. We only can generate a high number of test cases and look for anomalies. Of course the best result is if no anomalies are found in a high number of test cases.

This activity is in preparation. Therefore we could only mention test goals and results which we can predict from our current experience. However, in case of platform porting and subprogram testing it happened that we observed cases we never would have expected before. This may occur for this exercise, too.

5 CONCLUSIONS

The goal of this paper is to demonstrate the feasibility of generating independent software automatically. This has been proven for code and test case generation. Once a formal specification is available, generators can produce another version very fast and at low costs. It is even possible to check several generators – if available – against each other. If a formal specification is not available, but existing code, it may be derived from the code, so that even in this case an independent version can be created.

6 ACKNOWLEDGEMENTS

The work reported in this paper is supported by DLR (Deutsches Zentrum für Luft- und Raumfahrt) in the context of the General Support Technology Programme GSTP3 "Announcement of Opportunities", ESTEC contract no. 18670/04/NL/GC.

7 REFERENCES

[Aonix]	Aonix Ada compiler, http://www.aonix.com

[ARM95]	Ada95 Language Reference Manual, International Standard ISO/IEC 8652:1995(E), Lecture Notes in Computer Science 1246, Springer, 1997, e.g. http://www.ada95.ch/compiler/gnat.html
[ASIS]	Ada Semantic Interface Specification, http://www.adaic.org/standards/asis.html
[DARTT]	Dynamic Ada Random Test Tool, http://www.bsse.biz → Products → DARTT
[GNAT]	GNU Ada Compiler, AdaCore, http://www.gnat.com
[ISG]	ISG tool (Instantaneous System and Software Generation), http://www.bsse.biz → Products → ISG/ASaP
[MSC]	Message Sequence Charts, Z.120, ITU General Secretariat, Sales Section, Place de Nations, CH-1211 Geneva 20, Switzerland