Verification of the C++-Operating System RODOS in Context of a Small-Satellite

Evaluation of Software Robustness and Messaging Characteristics by Massive Stimulation

Rainer Gerlich, Ralf Gerlich

Dr. Rainer Gerlich BSSE System and Software Engineering
88090 Immenstaad, Germany
{Rainer.Gerlich Ralf.Gerlich}@bsse.biz

Karsten Gordon, Merlin Barschke

Institute for Aeronautics and Astronautics,
Technische Universität Berlin
10587 Berlin, Germany
{Karsten.Gordon Merlin. Barschke}@tu-berlin.de

Sergio Montenegro, Erik Dilger, Frank Flederer

Aerospace Information Technology
University of Wuerzburg
97074 Wuerzburg, Germany
{Sergio.Montenegro Erik.Dilger Frank.Flederer}@uni-wuerzburg.de

***Abstract*—Within the small satellite mission TechnoSat of Technische Universität Berlin, a verification strategy based on Dynamic Analysis has been applied to the C++-operating system RODOS using automated massive stimulation of the software-under-test. This approach is aiming at evaluating the robustness of the software and to derive feedback on the implemented messaging scheme of the on-board process chain. For fault detection and recording of message exchange the code is automatically instrumented with application-independent indicators which shall flag anomalies. Manual fault analysis is limited to the reported issues highlighting fault potential in contrast to usual reviews on the full code. The suggested reviews were extended to similar code, an approach which turned out as being effective. For the verification of the messaging scheme observed functional and performance properties were evaluated. The verification strategy targets the reduction of costs of verification and risks. Within this paper, the different verification steps are described and examples for reported issues are given.**

Keywords—software, verification, random testing, massive stimulation, fault identification, C++, small satellite

I. INTRODUCTION

A. General Overview

In the work described in this paper, massive stimulation is applied to the RODOS operating system (Real-time On-board Dependable Operating System) to analyze both its robustness and the characteristics of the messages exchanged within the processing chain. This analysis was started early in the development process to get a chance to improve the coding style due to the obtained feedback from already existing software for such parts which were still under development.

RODOS is used as operating system on the TechnoSat mission of Technische Universität Berlin [1]. It is a platform-independent framework for real-time software with an extension of message exchange based on the publisher-subscriber pattern, implemented in C++.

In comparison to C, C++ provides features favoring consistency and testability. However, it also requires fundamental changes in the test approach.

Massive, automated stimulation based on random input data is a method for dynamic analysis of a software system. In comparison with manual testing, it is able to exercise a much larger part of the input domain. Here, even very simple, generic approaches allow for verification of specific system properties in complex systems with a higher degree of confidence than manual testing activities can provide, because the high number of stimuli raises the activation probability of sporadic faults. A special variant of massive stimulation is fuzzing [2], where faults are identified by generic means such as looking for runtime exception or memory leaks. When combined with more specific checking means it can be used to verify individual properties of software.

Fault detection in the context of massive stimulation requires automatic measures for highlighting of faults or fault potential. Therefore, the code is automatically instrumented with indicators flagging anomalies like exceptions, and providing information on message exchange for verification of requirements. The instrumented code could be reused for verification of properties of the implemented messaging scheme. Cross-dependencies between independently observed data on communication were used for verification of both, the monitoring software and the software-under-test.

B. Definition of Terms

1) Coverage

The terms “block coverage” and “decision coverage” are used as a measure for test coverage expressed as ratio of the number of executed and total items (blocks, decisions).

A block consists of a sequence of one or more statements, where execution of the first implies execution of all other statements – provided that no exception occurs. A decision is a logical expression which may take one of two values, true or false, and both are considered for the coverage figure.

2) Fault, Error, Failure

A coding mistake (*fault*) may lead to an undesired internal state (*error*), which may become externally visible (*failure*).

3) Fault Injection

Fault injection is a technique by which a piece of software, e.g. a function, is exposed to invalid conditions in order to

check its behavior under such conditions, and more specifically – if robustness against faults is required – to check whether the injected fault does not manifest as a failure.

4) Fault Assessment

The definition of the terms "*false positive*", "*false negative*" and "*true positive*" are used in the context of fault report evaluation, indicating a report for a non-existing fault, a missed report for an existing fault, and an issued report for an existing fault.

As a decision on true or false positives requires manual analysis, the verification process may lead to effective false negatives due to a high number of reports issued by a verification tool, if not all reports can be processed manually within the allotted time and budget. Therefore a challenging goal of a verification tools is to minimize the number of false positives.

Although above definitions look quite clear, some ambiguity remains, namely in the definition of the actual classifier.

One degree of freedom is found in the *context* to be considered: Any component of a system can be considered on its own – *without context* –, where any component of the system may be exposed to the full input domain and robustness is checked. Alternatively, the component could be considered as part of a larger conglomerate of components – *with context* –, and only those defects and faults that can be activated within this larger structure should be reported.

Whether context should be considered or not during verification is a function of the goals of the verification process and the use of the system under verification.

In case of robustness testing the full range of inputs shall be considered and every anomaly found in the code should be considered as true positive. If anomalies are reported for inputs which never can occur in the given verification context, they may be considered as false positives, keeping in mind that this may invalidate the results of verification should the software ever be used within a different context.

For RODOS a mixed approach was applied. For components which are only used inside RODOS the context was considered leading to a constrained input domain with mainly automatically derived constraints, in order to reduce the number of false positives w.r.t. the context. The other components were exposed to the full input domain.

C. Structure of the Paper

The following chapters will provide an overview over the TechnoSat mission and its context (Chapter II), describe the verification strategy (Chapter III) and the obtained results (Chapter IV), discuss the lessons learnt (Chapter V), while conclusions and an overview on future work are given at the end of the paper in Chapter VI.

II. Description of the Verification Context

A. The TechnoSat-Mission

Technische Universität Berlin has a history of space science missions of more than 25 years. Since the university's first satellite, TUBSAT-A, was launched in July 1991, a total number of sixteen satellites were brought successfully into orbit, while another five are under development.

The university's latest research program is TUBiX. This platform series is developed in two different scales to support satellites with an approximate mass of 10 kg (TUBiX10) and 20 kg (TUBiX20), respectively [3].

The TUBiX20 nano-satellite platform's design objective is to meet different LEO mission requirements. To achieve a high level of flexibility regarding diverging mission scenarios, a generic, single-failure tolerant system architecture has been developed. The key design considerations for this architecture are modularity, reuse and dependability [4].

TechnoSat is a mission for in-orbit demonstration of novel nano-satellite technology [5] and carries seven different payloads. Table I gives an overview over the main parameters of the TechnoSat mission. It is in successful operation now over more than seven months.

TABLE I: TECHNOSAT MISSION DEATAILS

Orbit	600 km SSO
Launch date	July 14th, 2017
Design lifetime	1 year
Spacecraft mass	20 kg
Spacecraft volume	465 x 465 x 305 mm
Attitude sensors	IC magnetometers, Sun sensors MEMS gyroscopes, Fiber optic rate sensors
Attitude actuators	Torque rods
Payloads	• a fluid dynamic actuator (FDA) • an S-band transmitter (HISPICO) • fourteen laser ranging retro reflectors • a particle detector (SOLID) • a star tracker (STELLA) • a reaction wheels system with four wheels • a CMOS camera

B. RODOS

The software framework and operating system RODOS enables the definition and usage of *Building Blocks* (BB). It supports a variety of platforms. Before being used for TechnoSat, RODOS was part of several successful aerospace missions [6][7].

A BB is a collection of tasks that are implemented in software or hardware and the application developer defines its interface for data exchange. For this, RODOS provides a communication middleware which implements the *publish-subscribe* pattern. The data exchange is transparent, i.e. the BBs do not have to know whether the data comes from or goes to another node in a network. Thus, there is no need to modify a BB if a communication partner moves to a different node.

Topics define communication channels by their ID and data type. BBs can publish data to topics, which are received by BBs that have subscribed to topics. If several BBs subscribe to the same topic, every BB will receive the data in parallel like a multicast transmission.

A hardware abstraction layer (HAL) encapsulates the high-level concepts (e.g. communication middleware) from the individual hardware aspects. RODOS provides an object-oriented interface written in C++. Due to the dependability requirements of aerospace only a subset of C++ is used. Use of dynamic memory is discouraged, all memory has to be

allocated at startup. Exceptions are not used, instead an error code is returned.

Table II provides information about the size of the analyzed software, consisting of the used parts of RODOS and some part of the TechnoSat software required for evaluation of the messaging scheme. Table III gives some figures on telecommanding and on the number of topics and subscribers.

Table II: Overview of Software Size

	h	**cpp**	**Total**
Files	247	102	349
KLOC	33	15	48
KLines	60	22	82

struct	129
union	6
classes	142
functions	658

Table III: Processing Figures

Topics	16
Subscriber	42
Telecommand-Destinations	21
Possible Combinations Destinations / Telecommands	109

C. DCRTT

DCRTT (Dynamic C Random Test Tool) [8] is used in the context of verification activities based on its capabilities related to massive stimulation and reporting of issues in the code, and for analysis of the properties of the implemented messaging scheme based on the communication middleware.

In its most basic form it can be considered a generic fuzzing tool, which is applied on source-code level and automatically stimulates the possibly thousands of individual functions found within the source-code.

DCRTT completely automates the process between delivering the source code and reading the generated reports for thousands of functions under test, aiming to maximize variation of inputs in order to expose the software to extreme conditions which may occur only rarely under normal conditions.

A number of means have been implemented to increase the probability of fault activation and to detect an anomaly, to reduce the number of false positives and to provide comprehensive information. In addition, it generates test drivers for test vectors for regression testing, which are automatically selected according to coverage criteria or output from oracles (automatically) derived from semi-formal requirements.

False positives may only occur due to missing consideration of the context in contrast to static analysis where false positives may be a matter of the method, insufficient computing resources (time, memory).

1) Random Testing

Stimuli may be auto-generated randomly or grid-based from the valid or invalid input domain (black-box testing), or based on code analysis (white-box testing).

Constraints on ranges are derived automatically from the source code, as is information about the correlation of pointers with their associated length, to the degree possible.

The application code is instrumented for coverage, data range monitoring, rule-based checks and recording of runtime anomalies.

The approach shall complement other verification activities like functional testing / unit testing, static analysis and review. A comparison between DCRTT and static analyzers applied to middleware for space applications can be found in [9] for C, and in [10] for C++ where the Linux-version of RODOS was subject of evaluation. The results presented there suggest that such verification tools are complementary regarding their fault detection capabilities. In case of C++ the static analyzers delivered weak results regarding faults and potential faults compared to C. According to tool vendors this might be a matter of the complexity of the code.

DCRTT supports C according to ISO9899:2011 [11] and a large subset of C++ code as per ISO14882:2011 standard [12].

Some language features of C++ require specific consideration in the context of testing in general and automated stimulation in particular: data hiding, type polymorphism and templates.

2) Analysis of Messaging Characteristics

Utilizing the standardized interface provided by the publisher-subscriber paradigm implemented in RODOS and used by the application software, telecommands (TC) were randomly generated and injected into the system. Using the already existing instrumentation for coverage measurement and recording of anomalies, the data flow was observed and analyzed.

Information about the structure of the telecommands and the types of their fields was defined in a formal manner and used for code generation by the development team. That information, stored in CSV files, was then also used to establish automatically the TC generator.

Instrumentation of the methods involved in message distribution allowed to trace and record the cascade of messages generated upon reception of the TC and to monitor the data transfer in the channels used by these messages. The amount of transferred data in bytes was measured as well as the amount of accepted and rejected topic messages at the subscribers.

Based on this, utilization of transfer channels could be estimated given a usage profile for the TCs. Several of the recorded data items are connected by inter-dependencies, which were used to verify the correctness of the obtained information, thereby supporting a self-check of the instrumentation itself.

The data was also used to check several hypotheses about the correctness of the application code.

D. C++

C++ provides additional features which may increase consistency and verifiability, e.g. due to stronger separation of concerns, which in turn can facilitate more extensive use of the principle of compositionality in verification. However, some of these features require a very fundamental change in the test approach relative to that used for imperative languages, some others mainly induce additional complexity for testing due to different use patterns.

Data hiding requires a shift from direct assignment of values to parameters and structure elements – as is possible in C – towards use of constructors and available methods for

covering the set of possible object states. Dynamic binding introduces similar complexity for testing.

III. Verification Strategy

In general, the applied verification strategy does not require specific knowledge about the application to a major part. In case of robustness testing, the occurrence of anomalies – detected by application- independent checks – highlights weakness and fault potential, but may also – via further analysis – allow identification of application-specific logical faults specifically.

In case of the evaluation of the properties of the messaging scheme, checks are applied which are fully application-independent and could thus be reused for other systems based on RODOS. In addition, application-specific checks were added which require some knowledge of the system configuration.

The strategy is based on massive stimulation – implying extended fault detection means – as a complement for other verification means – targeting a guided review of code for which issues have been highlighted.

A description of the applied verification approach follows in Sections III.A and III.B.

A. Development Guidelines and Verification Status of RODOS

1) Guidelines

To make source code less complex, the guidelines for software development for RODOS encourage to use Embedded C++ (EC++), a pure subset of the ISO/IEC 14882-1998C++ standard [13].

2) Verification Status

RODOS already has been developed before the software development for TechnoSat began. It has been used within several aerospace projects before, which have been verified as whole systems. For the use on TechnoSat, RODOS was slightly modified in various aspects.

B. Implementation of the Verification Strategy

Verification of the selected software puts the focus on

- identification of fault potential capable of compromising operations, especially in view of sporadic faults,
- confirmation of correct and complete handling of the commanding chain triggered by telecommands and expanded internally by the messaging scheme based on the "publish-subscribe" pattern, and
- identification of potential overhead due to this (broadcasting) approach.

1) Random Testing

Robustness testing was applied to all functions by generating data of the full input domain as spawned by the parameter types of a function, except for functions for which a limited context could either be extracted automatically or – in some cases – manually, thereby limiting stimulation to the valid domain, while still being able to trigger fault handling.

Most of the context constraints were identified automatically by DCRTT from the source code. Some constraints were added manually when a limited context was identified during analysis, which was still missing.

Identification of (potential) faults is performed by rule checking and monitoring of anomalies. Rule checking is a deterministic approach (in the sense that every violation will be detected and reported) based on already known fault types like out-of-range conditions.

It is complemented by monitoring of anomalies like exceptions, but also by more sophisticated means which the DCRTT test environment does provide. Such monitoring does not guarantee that every violation will be detected and reported. But the optimization of such means in the past yields a reasonable probability to detect and report fault types even not known yet.

In contrast to usual code reviews, the review of the code was limited to such parts which were highlighted by issued reports. The reports were analyzed manually, pre-classified according to their fault potential and discussed with the developers.

In a two-step approach, a preliminary and a final version of the application software was analyzed:

- the analysis of the early first step gave hints on potential improvements for those parts which were under development or to be developed,
- the analysis of the second step should conclude on the remaining fault potential of the final version.

As could be expected, the number of relevant true positives was lower for the second step.

2) Evaluation of the Messaging Scheme

Due to the publish-subscribe pattern the messaging scheme is dynamically defined at start-up time by registration of the subscribers at the topic manager. Topics are issued from ground as telecommands on top-level, internally as timer signals on every level and by subscribers, as every subscriber can again issue topics. Actually, a subscriber does not know which other subscribers will receive the messages it did issue as publisher, as it does not know about registration.

A subscriber may check the contents of a received topic message and decide whether it is of interest (accepted message) or not (rejected message).

This dynamic behavior raised the interest to analyze in more detail the message exchange and a possible performance overhead due to received, but non-processed / rejected messages.

The coverage instrumentation of DCRTT was extended to track the message flow inside the application. Most specifically, coverage recording was used to determine whether a message was accepted or rejected and to record the related amount of data transmitted and received.

The approach supports fault injection to provoke rejected messages by invalid combinations of a telecommand-topic and a subscriber, a wrong number of telecommand parameters, or modification (activation / deactivation) of the distribution of a topic via an external channel.

The topic manager was (manually) instrumented to record the amount of data and the addressed subscribers.

The messages were classified according to criteria such as:

- accepted / rejected
- forwarded via an internal (shared memory) or external (e.g. bus) channel.

Cross-dependencies between observed data were identified like "the number of accepted and rejected messages must be identical to the number of issued messages". Similarly the ratio publish / put for a topic should be identical with the number of subscribers for a topic – plus 1 for transfer to a gateway if activated. If not, deeper investigation is required.

Other application-independent verification criteria are (non-exhaustive list)

- all topics / messages which are issued internally by a subscriber are accepted at least once,
- every valid TC is always accepted,
- no exception was observed during processing of the injected (valid or invalid) TC.

In addition, performance properties were recorded, such as

- mean and variance of message lengths and related channel load,
- maximum length of a message,
- the ratio between externally injected messages and internally triggered messages,
- ratio between internal and external communication (shared memory vs. communication medium/channel),
- the number of possible communication paths related to a TC,
- the maximum length of such a path,
- the number of path elements.

Further, the net of observed paths was visualized to support detection of deviations by manual inspection from what is expected.

IV. Results

A. Robustness Evaluation

A block and decision coverage of about 80% was achieved for about 660 functions under test based on massive auto-stimulation. These coverage figures are the basis of the analyses of the issued reports. General considerations on the applied analysis approach are given in Sect. A.1). Typical examples of reported anomalies are provided in Sect. A.2).

1) Issue-driven Analysis

Analysis was driven by the issued reports. The related code was reviewed to understand why the report was issued. This required to identify an extended code fragment around the affected location. It happened often that more issues were identified this way – related to application-specific faults.

Compared to reviews not guided by observed anomalies the motivation is quite different: the source of the reported issue *must* be found and understood, while in unguided reviews critical issues may not be detected due to code complexity, and the – possibly huge – amount of code to be inspected. The faults provided in the following examples might have been detected in the course of unguided reviews, too. The essential question is, however, whether they would actually have been found. The same is true for static analyses, especially in view of the remarks in Sect. II.C.1).

According to obtained experience (in this and other exercises) the presence of an application-independent fault seems to suggest that application-dependent faults may be present, too, in the addressed code.

The time required to come to a conclusion on an issued report (true positive, false positive) may vary largely, from a few minutes to several hours, depending on the complexity of the context. The majority of reports required up to 15 minutes, 30 minutes and more were required sporadically. The higher duration is a matter of required deeper investigation, with re-runs and more sophisticated instrumentation. C++ code tends to require more time e.g. in case of polymorphism.

In some cases the pattern causing the anomaly occurred at different locations, so that results of previous analysis could be reused. It should be mentioned that these cases are different from the ones for which reports are repeated for the same location in the code, but reached by different paths. DCRTT filters such repeated reports to reduce analysis time. However, all reports are still available, if access is required.

Before an analysis result is derived, no information on the fault potential is available. Therefore no report can be excluded a priori. It was beneficial to extend analysis to code which was actually not subject of testing.

Considering, for example, the following faulty index check

`var<upper_bound`

while the correct coding should be

`var≤upper_bound`

It occurred because `upper_bound` was already decreased by 1 immediately after the related value on the maximum number of elements was assigned – in contrast to usual conventions. Therefore it could be assumed that there might be more of such wrong comparisons and the code was checked for other such faulty occurrences irrespectively whether the related code in the RODOS library was actually subject of testing and more locations were found.

Another example is faulty code in equivalent functions in subclasses, e.g. one function is in a class for Platform A, and the other is in a class for Platform B, but both are called by the same – platform-independent – caller, evaluating the return code. In case an issue related to the return code (mix signed/unsigned, see Sect. IV.A.2)c)) was detected, all similar callees were checked, even if only A is under test, but not B.

In one case -2 was returned, in the other case 0. This resulted in different behavior: in one case a loop with 2^{32}-2 iterations (0xfffffffe for -2) would have been executed, in the other case the loop would be not executed – which is the desired behavior.

2) Typical Examples

The essential point regarding the given examples is not the fault type, but the fact that issues were reported and the fault potential could be assessed.

a) Invalid Access

Typical examples are index-out-of-range and invalid pointer.

Out-of-range may have several sources (non-exhaustive list):

- -1 is returned as error indicator, but propagation of this value is not prevented due to missing error handling, and could be interpreted as unsigned. Faults of this kind were detected by fault injection.
- A pointer to a shorter structure is expected as parameter and is casted to a pointer of a longer structure inside the function. The cast hides the conflict and prevents the compiler flagging of the type / length incompatibility. Passing a structure which complies with the prototype size resulted in an invalid access detected by the checking facility of DCRTT. In the TechnoSat context a structure of sufficient length is always passed. Therefore, no fault was observed so far.

b) Volatile Data

When compared to functions, macros have the advantage that a dedicated consideration of types is not required like in case of min/max:

```
#define MY_MIN(x,y) (x<y?x:y)
```

It does it for any type.

However, in case of volatile data, the values may change, and the code is not safe from a rigorous point of view (although this might occur very sporadically, only).

This impact was detected by using the macro with random data on every occurrence of the arguments x and y.

c) Mix signed / unsigned

A mix of signed and unsigned types in expressions was frequently observed, implying either explicit or implicit conversions. The potential impact of faults in such conversions is high, although not every such fault may directly manifest as a failure in the given context.

Typically, such faults were observed in context of negative return as error code. Due to stimulation the error handling code was executed, while not under normal operation.

```
unsigned int len=
   MIN((int)maxLen,getLenDest(index));
len=MIN(len,    getLenSrc(source));
memcpy(dest[index],src,len);
```

maxLen is of type *unsigned* and provides the maximum length of `source`, *getLenDest* returns the maximum length of the destination as *int*, *getLenSrc* returns the actual length of the source as unsigned, MIN is the macro mentioned in b) above.

In case of an invalid *index getLenDest* returns -1, which means the result of MIN is -1 as the comparison is done on *signed int*. So *len* gets the value -1 as unsigned int due to implicit conversion, which is 0xffffffff or 2^{32}-1. In the second step the comparison is done on *unsigned int*, and *len* gets the return value from *getLenSrc*. This means, not the wrong value 0xffffffff propagates but the current length of the source is taken for the following operation. As long as

```
len < length(dest)
```

holds, the intended portion will be copied: the operation will complete as desired – although a completely wrong result occurred intermediately.

However, the goal to consider the minimum of *len*, too, was not achieved. As the size of the source exceeded the size of the destination due to random stimulation, an anomaly was reported by DCRTT.

It seems that so far *getLenDest* never returned -1, or if it returned -1, the *memcpy* never caused an undesired state, as the constraint on *len* was fulfilled.

d) Identificaton of Potential Inconsistencies

Stimulation following the – unbiased – information obtained from the prototype can identify potential inconsistencies due to inherent dependencies / assumptions which are not reflected in the code. E.g., the function

```
void myFunc(int *arr) { int i; for (i=0;i<10;i++)
                    arr[i]=0;}
```

hides the dependency on the size of arr: exactly or at least 10 elements are expected. If the context is changed, e.g. in the course of maintenance, the potential fault could be activated. Stimulation with randomly chosen size did raise an issue.

e) Edge Cases

Due to stimulation over the full input domain, faults at edge cases were activated.

An example is the loss of a bit in function *setField* which shall set a bit field in a char* stream for which the bit position in the stream and the field width in bits are the parameters. The algorithm was only correct for

```
bitPosition mod 16 != 0
```

and

```
0 ≤((bitPosition mod 16) + fieldLength) ≤ 15
```

This case was detected due to random stimulation using the full range of the inputs. Further, the analysis yielded that a negative argument for a shift operation could occur, which is an undefined operation according to the C standard, but yielded the correct result for the actual context.

A check of the RODOS code for calls of this function (and the associated *getField* function with similar issues) yielded that the critical values were not used.

f) Imperfect Fault Handling

In a number of cases either a fault in a fault handling part or incomplete fault handling was observed due to injection of invalid data.

An example for first case was already given in Sect. 1) above for the check on `upper_bound`.

Due to the check on a wrong upper bound one erroneous cases cannot be detected.

In other cases fault handling was incomplete because either checks were not implemented and fault propagation could occur, or were implemented but e.g. a fault is masked by replacing an invalid by a valid value, but the occurrence of the fault is not flagged. In this case the chosen valid value may not prevent fault propagation, as the assigned value may still be wrong, too, and further, the source of the fault cannot be identified and fixed.

B. Analysis of Messaging Characteristics

Like for robustness testing, also for this verification step two iterations were performed for an early and a late version.

As the software related to this part of verification was mainly developed in addition to the existing software for

robustness testing, initially, faults were found in this new and additional code due to the cross-checks as described in Section III.B.2), supporting fault detection not only in the software-under-test, but also in the verification support software

The stability of the observed results with statistical variations was checked. A number of runs were performed at an increasing number of stimuli, and their convergence and compliance of the deviations with the theoretical statistical limits was confirmed.

- The results for the messaging scheme obtained during the second iteration were compliant with the rules of Sect. III.B.2), except e.g. when a wrong number of telecommand (TC) parameters was identified.
  It turned out, that correct contents is expected and must be guaranteed by ground and uplink.
- All messages sent internally are accepted.
  The "Anomaly Reporting" facility did not process messages with lower criticality, and rejected / dropped them.
- All valid TCs are always accepted.
  Due to the publisher-subscriber broadcasting of messages, TCs were received at subscribers which were not really intended as receivers. Such messages could be considered as an overhead due to the publish-subscribe pattern.

In two cases the ratio publish / put was not an integer number, neither identical with the number of subscribers or subscribers + 1 (for the gateway). The analysis yielded that not all subscribers issuing the put for the considered topic had activated a transfer via the gateway. In another case it was detected that the pair (topic, subscriber) was not unique – unintendedly so.

The performance analysis yielded that the overhead due to broadcasting and rejected messages is sufficiently low.

The observed paths of communication were checked on the base of the graphical figures and their correctness was confirmed.

In the following cases deviations from the expected figures were observed:

- Subpaths were missing due to object files which were unintentionally not linked into the executable.
- Some topics were missing because the stimulation weight was unintentionally set to 0.
- The ratio of issued topics to received topics was not as expected, as parallel sending via an external channel was disabled due to a specific test condition, unintentionally left in the code.

Cases 1 and 2 were a matter of conditioning of the test environment, while Case 3 was related to the code to be verified.

These observations confirm that deviations from the expected behavior in the messaging scheme can be identified with the implemented checks and visualization means, and can detect faults in the code to be verified, and in the verification environment itself, too.

V. Lessons Learned

Issues were raised regarding language, application and verification.

A. Language Issues

1) C and C++

The issues listed here apply to C and C++.

a) Mix signed / unsigned

In Section IV.A.2) c) the fault potential of a mix of signed and unsigned types was discussed. In case of the abs-library function such a mix is enforced by its return type being a signed integer:

```
int abs(int)
```

This raised an anomaly for the edge case *INT_MIN*, because a following check on a positive limit failed assuming that the return value would always be positive.

According to both the C- and the C++-standard [11][12] the behavior is undefined if the result cannot be represented as *int*, which is usually the case for *INT_MIN*, as *–INT_MIN* is typically larger than *INT_MAX*. It is quite plausible to expect the *abs* function to return an unsigned integer, thus leading to a conflict between intuitive and correct use of the function.

Although this edge case might occur very rarely only, it is a principal issue from a safety point of view.

b) Arrays as Pointers

For arrays which are passed as parameter to a function, the information on the number of elements is lost for the left-most dimension, even if all dimensions are explicitly provided in the prototype:

```
void myFunc(int arr[10]);
```

is interpreted as

```
void myFunc(int *arr);
```

In consequence, no information on the array size is available inside the function at compile-time or at run-time.

2) C++

In C++, declaration of a variable with object type always implies the invocation of an associated constructor. For global variables these invocations happen at program startup, before invocation of the main function [12]. The order in which the objects are initialized is only partially specified: The objects within a single compilation unit – i.e. a single source code file – are initialized in the order of declaration. However, the order of initialization between different compilation units is not defined by the standard.

For embedded applications the order of initialization may be of utmost importance. For example, objects representing hardware drivers need to be initialized before the objects using them.

Therefore, initialization in RODOS happens within special methods which are invoked from the main function of the application. This deviates from best practices in that the objects are not actually in a useable state immediately after construction.

This also impacts automatic testing, as standard mechanisms of construction are not sufficient for test data generation, and proper solutions had to be added.

B. Application

1) Resource Management

In embedded applications, dynamic memory management usually is not applied. Instead, all resources are allocated at

startup and never freed. De facto this makes definition of destructors unnecessary, and thus many if not most classes in RODOS and the application do not have destructors.

This, however, poses a problem to automatic testing, as objects are not destroyed after test execution, causing consumption of more and more resources that cannot be freed. The only alternative would be to restart the test executable for every test case to have a clean plate every time.

2) Casts

Casts compromise the checking capabilities of a compiler. Resulting faults can only be detected at run-time if the required means are provided like tracking of object lengths of heap, stack and memory allocated with malloc, as DCRTT does.

C. Verification

The fault types described above usually are occurring (very) rarely, because the probability of fault activation is very low, e.g. for fault handling code, or they cannot occur in the current context, but could when the context is changed. Massive stimulation over the valid and/or invalid input domain can activate such faults

Several cases were found showing that (intermediate) results may be wrong from a logical point of view (see c) and e) in Ch. IV), but this may not necessarily cause a failure. Whether a fault will manifest depends on the context. To know about such fault potential is essential regarding risk reduction and fault avoidance. Getting such information early avoids their duplication during development. This reduces the analysis effort and the costs of verification.

VI. Conclusions and Future Work

Due to automated generation of the environment for testing, massive stimulation could be applied to about 660 C++ functions without manual intervention during test setup and execution. It raised several issues in the code and gave answers on a unit's behavior under stress conditions and on the characteristics of the actually implemented messaging scheme for several operational profiles. The obtained knowledge was either considered for improvement of code or confirmation of expected functionality.

Our verification strategy applied code reviews guided by the issued reports. The knowledge about existence of an issue increases the probability of fault detection in a limited piece of code compared to unguided reviews, e.g. on the whole code.

The experience with two iterations confirms that it is beneficial to start early with verification of code – as soon as code is available. This is strongly recommended to reduce the total verification effort by avoiding duplication of critical issues.

Robustness testing of functions highlights valuable issues, but may also point to issues not relevant for the current operational context. Their relevance can only be shown after manual analysis of a raised issue. Taking the view of the operational context, the effort for irrelevant issues seems to be an overhead. From a rigorous safety point of view it is acceptable, and the identification of potential weakness is desired. However, a benefit is the future avoidance of the potentially irrelevant issues in the course of further development.

Functional faults were detected in algorithms due to issues raised by application-independent checks on edge cases. Issues related to C and C++ language were identified in context of the reported anomalies.

Future work shall extend the automated approach towards application-driven stimulation and checking, e.g. to derive oracles from (semi-formal) requirements.

Acknowledgment

The TechnoSat mission is funded by the Federal Ministry for Economic Affairs and Energy (BMWi) through the German Aerospace Center (DLR) on the basis of a decision of the German Bundestag (Grant No. 50 RM 1219).

References

[1] M.F. Barschke, K. Gordon, M. Lehmann and K. Brieß, "The TechnoSat mission for on-orbit technology demonstration", presented at the 65th German Aerospace Congress, Braunschweig, Germany, 2016.

[2] Miller, B. P.; Fredriksen, L. & So, B., "An Empirical Study of the Reliability of UNIX Utilities", Communications of the ACM, Vol. 33, No. 12, pp. 32-44, 1990.

[3] M.F. Barschke, Z. Yoon and K. Brieß, "TUBiX – The TU Berlin innovative next generation nanosatellite bus", presented at the 64th International Astronautical Congress, Beijing, China, 2013.

[4] M.F. Barschke and K. Gordon, "A generic systems architecture for a single failure tolerant nanosatellite platform", presented at the 65th International Astronautical Congress, Toronto, Canada, 2014.

[5] M.F. Barschke, K. Großekatthöfer and S. Montenegro, "Implementation of a nanosatellite on-board software based on building-blocks", presented at the Small Satellites Systems and Services Symposium, Majorca, Spain, 2014.

[6] S. Montenegro, "RODOS operating system for Network Centric Core Avionics", Conference on Advances in Satellite and Space Communications 2009

[7] Thomas Walter, Alexander Hilgarth, Tobias Mikschl, Sergio Montenegro, "VIDANA: A fault tolerant approach for a distributed data management system in nano-satellites", 10th Symposium on Small Satellites for Earth Observation 2013

[8] R. Gerlich, R. Gerlich, M. Prochazka, K. Kvinnesland, B. Johansen, "A Case Study on Automated Source-Code-Based Testing Methods", Eurospace Symposium DASIA'2013 "DAta Systems in Aerospace", May 14th-16th, 2013, Porto, Portugal

[9] R. Gerlich, R. Gerlich, A. Fischer, M. Pinto, C. Prause. Early results from characterizing verification tools through coding error candidates reported in space flight software. DASIA, 2016.

[10] R. Gerlich, R. Gerlich, S. Montenegro, F. Flederer, J. Gerlach, J. Burghardt, C. Prause. Evaluation of verification tools continued: More tools, more software, more aspects. DASIA, 2017.

[11] International Standard ISO 9899:2011: Information Technology – Programming Languages – C, 2011.

[12] International Standard ISO 14882:2011: Information Technology – Programming Languages – C++, Third Edition, 2011.

[13] The Embedded C++ specification, http://www.caravan.net/ec2plus/spec.html