A metamorphic testing approach to NASA GMSEC's flexible publish and subscribe functionality

Julian Rothermel
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
jrothermel@fc-md.umd.edu

Mikael Lindvall
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
mikli@fc-md.umd.edu

Adam Porter
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
aporter@fc-md.umd.edu

Sigurthor Bjorgvinsson
Fraunhofer CESE
5825 Univ. Research Ct
College Park, Maryland
sbjorgvinsson@fc-md.umd.edu

ABSTRACT

Today's software systems are increasingly required to be flexible which is achieved by providing various forms of loose coupling and configuration options. While loose coupling and configuration options facilitate quick adaptation to changing requirements, such flexibility increases the difficulty of system testing. It is often relatively straight-forward to create different configuration options as test cases, but it is typically much more difficult to formulate the expected system behavior, which is known as the oracle problem. NASA's GMSEC software bus is such a flexible system that serves as a central communication channel for software components based on a publish and subscribe architecture where several software components can be dynamically connected to the system. To cope with the difficulties in testing such a flexible software system, we present a metamorphic testing approach that explicitly addresses the test oracle problem. In this paper, we focus on testing the publish and subscribe functionality of GMSEC motivated by the fact that its middleware-based system architecture is the foundation of many of NASA's missions.

KEYWORDS

Test case generation; middleware systems; metamorphic testing; pub-sub architectures.

INTRODUCTION

Today's software systems are increasingly required to be flexible and achieve flexibility by providing various forms of loose coupling and configuration options. NASA's Goddard Mission Service Evolution Center (GMSEC) is a case in point. One component of GMSEC is a software bus that serves as a central communication channel for several other software components of GMSEC. The bus is based on a publish and subscribe architecture where several software components can be dynamically connected to the system. In the publisher-subscriber (pub-sub) architectural style, different components communicate in an indirect fashion by publishing and subscribing to messages managed by an intermediate communication bus (or broker) [17]. This indirect communication makes the architecture flexible because it facilitates adding and removing software components.

However, this increased flexibility of the publisher-subscriber architectural style makes it difficult to predict the emerging behavior and to demonstrate the correctness of the integrated system even though each individual component passed testing. Thus, while flexibility increases, analyzability and testability decreases [2].

One example of the decreased testability is that it can be difficult to formulate an oracle that determines whether or not messages published to the bus were received as expected. If defects in the interpretation of subscriptions remain in the deployed system, important messages can be lost with potentially disastrous effects. In addition, if sensitive messages are incorrectly distributed to clients who are not permitted to see them, security breaches may result.

In this paper, we focus on testing the publish and subscribe functionality of GMSEC motivated by the fact that its middleware-based system architecture is the foundation of many mission-critical projects at NASA. GMSEC, which is a mature software system, has been tested thoroughly over many years and is successfully used in many NASA missions [18]. As a result, it has high quality and is a robust system and is now provided as open source, which is the version we tested. Nevertheless, by applying our proposed testing approach, which is based on the principles of metamorphic testing [19], to GMSEC, we revealed new, previously unknown issues. The issues have been reported to the GMSEC team.

GMSEC

GMSEC is a reusable framework designed as a product line that allows missions to build new ground systems in short time. Being built around a software bus that is in charge of forwarding messages, applications can be easily plugged in to and out from the software bus without software changes. Designed to be the basis for ground systems, GMSEC provides a wide range of different system configurations along with standardized interfaces. Software developers can develop and connect their own applications to the software bus and can use existing applications [16], [18]. The *GMSEC API* is used to communicate with the software bus and

ACM ISBN 978-1-4503-5729-6/18/05…$15.00
https://doi.org/10.1145/3193977.3193984

provides a multi-language and cross-platform interface allowing for seamless integration of applications written in different programming languages. The software bus is implemented by a replaceable middleware which is accessible via a wrapper, which allows the middleware implementation to be replaced by an alternative implementation without change to plugged-in software components.

In the past, we analyzed and tested earlier versions of GMSEC by using static and dynamic analysis [2]. We concluded that GMSEC, in addition to having high quality, has a good middleware abstraction layer, which helps in avoiding vendor lock-in. We also detected some high-priority bugs due to behavioral discrepancies among different middleware wrapper implementations [2]. We used model-based testing to test GMSEC's behavior for different programming languages as well as for different middlewares [4], [16]. We found that even though the behavior should be identical, independent of the programming language and different middlewares, this was not always the case. Many of the detected issues have since been corrected by the GMSEC team. In this paper we continue testing of some of the most important functions of GMSEC, namely the pub-sub functionality, which we did not test in detail before. More specifically, we never tested subscription-oriented commands such as the *excludeSubject* and the *removeExcludedSubject* commands. Another difference is that we have not used metamorphic testing to test GMSEC before. Instead we used model-based testing to generate test cases. Each test case had assert statements built in, thus avoiding the oracle problem. This was not an option for in-depth testing of the subscribe function since it requires a relatively extensive apparatus to keep track of exactly what subjects should be accepted or rejected. Lastly, the issues reported in this paper are new and have not been reported before.

The Message Bus

In the pub-sub style, there are four key concepts in addition to the software bus: *publishers*, *subscribers*, *messages*, and *subjects* (a.k.a. topics). *Subscribers* subscribe to one or more subjects. *Publishers* publish messages related to one or more subjects. A subscription can specify the subject as a pattern using a restricted form of regular expressions. When a message belonging to a certain subject has been published, the bus delivers the message to the subscribers whose subscriptions match the subject. Thus, subscribers and publishers are loosely coupled and unaware of each other; they are only aware of subjects and messages. Each application connected to the bus can be a publisher or a subscriber, or both, allowing an application to receive messages from other applications (as well as from itself) without knowing who sent them, unless sender information is included in the message. In addition to a subject, each message has content, i.e. the fields and values representing the data to be sent. Each subscriber periodically checks if there is a message waiting. If so, the bus delivers the next available message when the subscribing application so requests. See detailed examples in [2]. There are a few rules for the bus that are important for testing: 1. Delivery of messages: In order for a message to be delivered, an application must subscribe to the subject before messages of the same subject are published. Only if this is the case, the published message is delivered to the subscriber's buffer. Once the message is delivered to the buffer, the subscriber can retrieve that message at any time. 2. Reliability: The bus should deliver one copy of each message to each subscriber, once and only once.

GMSEC API

Following a standardized pub-sub paradigm, GMSEC provides an API that offers a multi-language standard communication interface. Our work presented in this paper exclusively considers the GMSEC API for Java. Throughout our testing process we use the GMSEC API version 4.4. Regarding the subscription to message subjects, the GMSEC API offers four main functionalities on which we will focus in the following.

- *subscribe(...)*

Subscribes to a particular subject given as an argument. The subject is expressed as a string or a pattern. Duplicate subscriptions are not possible to the effect that an exception is thrown if the subscribe method is called twice with the same subject and without an unsubscribe invocation for the particular subject in between.

- *unsubscribe(...)*

Unsubscribes to a particular subject. A successful execution of the unsubscribe method will stop the receipt of messages whose subjects match this particular subject. Given a subject *x*, this method throws an exception if a subscribe method invocation for that particular subject is missing before this method invocation. Thus, the order is important: *unsubscribe(x)* must be preceded by *subscribe(x)*. A successful unsubscription to a subject removes the subscription for that subject allowing the application to subscribe to the same subject later.

- *excludeSubject(...)*

Excludes incoming messages whose subjects match the specified subject. The subject is a string or pattern.

- *removeExcludedSubject(...)*

Removes an excluded subject allowing incoming messages with matching subject to be received again.

With the exception of *unsubscribe(...)* all methods take a string object as the argument. In contrast, *unsubscribe(...)* takes a *SubscriptionInfo* object as the argument. Such an object is returned by a successful execution of *subscribe(...)*. This construction ensures that we only unsubscribe from subjects that we subscribed to before. Subjects must follow certain syntax rules [7]: A subject has one or more element that are separated by a dot. Figure 1 illustrates a subject with five elements.

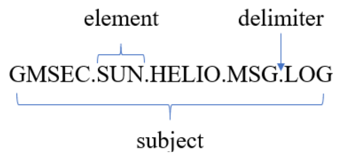

Figure 1. Subject consisting of five elements separated by four delimiters.

The documentation of the GMSEC API [7] does not specify any limit on the number of elements within a subject but suggests ensuring that the number does not become extremely high due to possible loss of performance. Regarding the length of the elements the documentation proposes not to exceed 12 characters, however, a limit is not set by the GMSEC API. The specific element names can be chosen by the programmer and are not predefined by the GMSEC API, but certain restrictions must be considered. A subject's character set includes all upper case letters of the alphabet (*A - Z*), digits (*0 – 9*), the special characters _ (underscore) and – (hyphen). Additionally, three characters are introduced that can only be used by the methods *subscribe(…)*, *excludeSubject(…)*, and *removeExcludedSubject(…)*.

The *asterisk* (*) serves as a wildcard and matches a complete element. It can be used at any position within the subject. Let's consider the following example in Figure 2 where a client system subscribes to a subject *"GMSEC.SUN.HELIO.*"*.

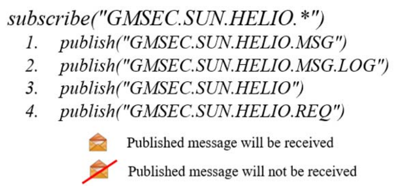
subscribe("GMSEC.SUN.HELIO.*")
1. publish("GMSEC.SUN.HELIO.MSG")
2. publish("GMSEC.SUN.HELIO.MSG.LOG")
3. publish("GMSEC.SUN.HELIO")
4. publish("GMSEC.SUN.HELIO.REQ")
Published message will be received
Published message will not be received

Figure 2. An example of published messages and their reception at the subscriber based on the subscription subject.

To receive a message, the subject of the published message must have exactly four elements where the first, second and third element must be identical to the specified elements. The fourth element can be any character combination that fulfills the rules of the GMSEC API. A subject with more or fewer than four elements does not match the subject of the subscription.

The *greater than* (>) character is only allowed to be placed at the right-most position of a subject and must be directly appended to the delimiter or alone (e.g. *subscribe("">"*)). A match with the subject of published messages is achieved only if at least one element is appended to the initial string of the published message's subject. For example, *"GMSEC.SUN.HELIO.>"* means that subjects must have four or more elements and exactly match the first three elements.

Similar to the *greater than* character, the *plus* (+) character must be placed right after a delimiter and only at the right-most position of a subject or alone (e.g. *subscribe("+")*). The *plus* character extends the functionality of the *greater than* character such that a match with the subject of a published message is already achieved if no element is appended to the initial subject string of the published message. Other than that, the functionality of the *plus* operator corresponds to the one of the *greater than* operator. For example, *"GMSEC.SUN.HELIO.+"* means that subjects must have three or more elements in addition to exactly matching the first three elements.

The example in Figure 3 shows how two subscribers and two publishers communicate with the software bus and how it distributes published messages to subscribers whose subscriptions match the subject of the published messages. Subscriber1 starts by subscribing to the subject *"A.* "*, but immediately excludes subject *"A.A"*. Thus, any message for which the subject starts with *"A"*, has two elements, but doesn't end with *"A"*, will be delivered to Subscriber1. Subscriber2 also subscribes to *"A.*"*. Publisher1 publishes a message with the subject *"A.A"*. This message gets delivered to Subscriber2 but not to Subscriber1. Publisher1 publishes a message with the subject *"A.B"*. This message gets delivered to Subscriber1 and Subscriber2 because it matches both subscribers' subscriptions. Subscriber2 then unsubscribes to *"A.*"* while Subscriber1 removes the excluded subject. When Publisher2 publishes a message with subject *"A.A"* the effect is that Subscriber1 receives the message, but Subscriber2 does not. This example illustrates how different sequences of commands can achieve equivalent results: *subscribe("A.*")* followed by *excludeSubject("A.A")*, followed by *removeExcludedSubject("A.A")* is equivalent to *subscribe("A.*")*. The example also illustrates that it does not matter who publishes the message: if the subscription matches the subject of the published message, the message is delivered.

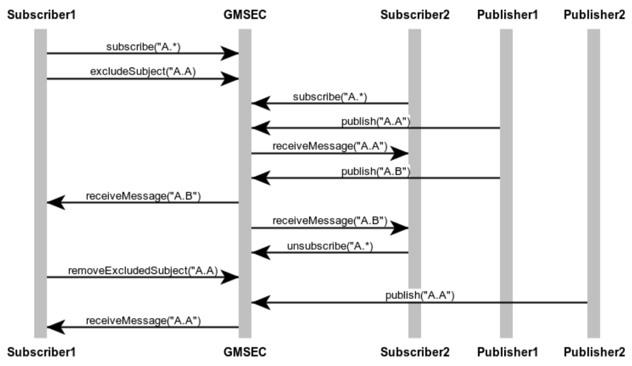

Figure 3. A sequence that illustrates the use of the four commands tested in this paper using two subscribers and two publishers.

METAMORPHIC TESTING OF THE GMSEC API

The subscribe and publish functionality of NASA's GMSEC software system architecture and its quantifiers (*, >, and +) make it possible to create complex subscriptions. Therefore, it becomes difficult to verify the correct functioning of possible subscribe, unsubscribe, exclude and remove exclusion combinations in the context of the receipt of published messages. The difficulties, which arise when manually defining the expected test result in terms of receiving a published message, are alleviated by using a metamorphic testing approach. In this approach an initial test case, called *source test case*, is manually created and serves as a basis for additional so-called *follow-up test cases*.

The relation between a source test case and its follow-up test cases is defined by a *metamorphic relation*. Such a relation can have different properties which describe the relation between the expected result of the source test case and its follow-up test cases [20], [21]. For example, this could be the equality of the test case results. If the result of a follow-up test case corresponds exactly to the result of its source test case, the follow-up test case passes, otherwise it fails. Instead of constructing complex test case oracles which tend to be expensive to create, metamorphic testing reuses test results from the source test case for generated follow-up test cases based on the previously defined metamorphic relations.

In applying metamorphic testing on the publish and subscribe functionality of NASA's GMSEC software bus, we deduct general rules based on the documentation of the GMSEC API and the offered functionalities described above. These rules form the basis for a definition of metamorphic relations between different statement combinations. Statement combinations comprise all functionalities offered by the API: subscribing to a subject (*subscribe(...)*), unsubscribing from a subject (un*subscribe(...)*), excluding a subject from a subscription (*excludeSubject(...)*), and removing the exclusion of a subject (*removeExcludedSubject(...)*). Further, our metamorphic relations explicitly consider a subject's composition, i.e. its elements and their composition, and the element positions within the subject.

Classification of our testing approach

We identified two classes of metamorphic rules. We call the first class of metamorphic rules *True equivalences* and assign metamorphic rules to this class if the equivalence of statement sequences of different client systems is ensured for all possible subjects of published messages. The second class refers to a testing approach which is applied on subscribing client systems which only have a *True equivalence for a given set of subjects.* This means that two statement sequences of different client systems can result in different messages being received. In that way, one client system receives a message with a particular subject while the other client system does not receive the message. For example, a client system which subscribes to *"*"* via *subscribe("*")* is only equivalent to a client system which subscribes to "A" via *subscribe("A")* if the subjects of published messages are restricted to the subject *"A"*. Therefore, testing of these statement sequences together can only follow the procedure of the second class. Here, both client systems receive the published message with subject *"A"*. However, if a published message would have a subject *"B"* only the client system with *subscribe("*")* receives the message. Thus, for such equivalences the subjects of the published messages must be constructed in a way that makes the relationship truly metamorphic, i.e. that the subscribers receive the same messages. Our procedure of creating tests cases and the applied rules that belong to the class of *True equivalence* are described in the following sections as well as the procedure for *True equivalence for a given set of subjects.*

Metamorphic Testing for True Equivalences

In the first step of testing the GMSEC API we manually create so-called statement sequences. In doing so, we thoroughly analyzed the documentation of the GMSEC API to be able to deduce statement sequences which follow its sequence and subject rules. Such statement sequences can comprise invocations of all four functions offered by the bus, but must not contain each of the functions. Since these statement sequences are an essential part of our source test cases we create them manually while explicitly considering the offered functions and potential vulnerabilities of the bus. For each subject, we consider the complete set of possible characters. Based on the source test cases we automatically generate follow-up test cases which follow predefined rules, see next section.
As a last step of the test case generation we generate a set of subjects for messages to be published similar to a model-based test case generation. In doing so, we first programmatically analyze all test cases (source as well as follow-up test cases) and generate a number of subject strings that match at least one subject of all subscriptions given in the statement sequences of all test cases. We then generate a set of random subject strings for messages to be published without considering the subscription statements.

Metamorphic Rules

In the following, we describe the metamorphic rules applied in our testing approach. Being aware of different output relationships of the test cases that make use of different metamorphic rules [20], we exclusively focus on metamorphic relations that follow a pattern which we call *Equivalence* where the output of the source and its follow-up test cases must be equal, i.e., the subscribing side of the source and its follow-up test cases must receive exactly the same published messages in the same order. Other output relationships known from the literature, such as *Sets*, same items within set, but not necessarily same order, Disjoint, Subsets (see [20] and [21] for example) are not in the scope of this paper.

1) Substitution of quantifier:

Since the character ">" (greater than) matches one or more elements to the right, it can be replaced by the characters *"*.+"*: The "*" (asterisk) ensures that exactly one element is matched while the appended "+" (plus) ensures that it takes place of zero or more elements. Using them in combination ensures that this expression matches at least one element which corresponds to the meaning of ">". For example, a client system that subscribes to messages which match the subject *"A.>"* (*subscribe("A.>")*) should receive exactly the same messages as a client system which subscribes to topics which match *"A.*.+"* (*subscribe("A.*.+")*). Similarly, *"*.+"* can be replaced by the character *">"*.

2) Order of subscriptions and exclusions:

Given a statement sequence which contains occurrences of a subscription and exclude method, then their mutual order should not matter. That is, the GMSEC API does not complain if an exclude command has been issued without previously submitting a subscribe command. For example, the following two statement sequences which represent two subscribing client systems that interact independently from each other, receive the same messages.

1. *subscribe("A.*"),*
   excludeSubject("A.A")
2. *excludeSubject("A.A"),*
   subscribe("A.")*

If a message with subject *"A.B"* is published, both clients receive the message while a message with subject *"A.A"* will not be received by any of the two clients.

3) Redundant statements:

Adding redundant subscription statements should have no effect regarding the change of a result, i.e. the receipt of a message. Given the sequence *subscribe("*"), excludeSubject("A").* When adding a redundant statement *subscribe("A")*, the subscriber should still receive all messages with a length of one element, except a message with subject *"A"*. Since *"*"* serves as a wildcard which takes place of exactly one element of a subject, the subscription to subject *"A"* is already ensured.

4) Additional subscriptions nullified by unsubscriptions:

Adding additional subscriptions to a statement sequence which are nullified afterwards by adding unsubscriptions from the corresponding subject should not result in any changes of the results regarding the receipt of messages. Considering for example the statement sequence *subscribe("A")*. By adding the two nullifying statements *subscribe("B")* and *unsubscribe("B")*, the respective subscribing client system should always receive only published messages with a subject *"A"*. For example, a message with subject *"B"* should not be received. Please note that we identified three positions where adding these nullifying statement sequence combinations of *subscribe()* and *unsubscribe()* are suitable: at the beginning of the statement sequence of the test case, at the end of the statement sequence of the test case, or at a random location within the statement sequence of the test case. The only requirement is that all sequences that together form a nullifying statement sequence must be placed together.

5) Additional exclusions nullified by removals of exclusions:

The following modification should not result in any changes of the results regarding the receipt of published messages: Adding exclusions of particular subjects which are nullified afterwards by adding statements that remove the previously introduced exclusions. For example, given the following statement sequence *subscribe("A")*. A client system that subscribed only to messages with subjects *"A"* should only receive messages with subject *"A"*. Other messages should not be received. Adding the statement *excludeSubject("A")* which is followed by a nullifying statement *removeExcludedSubject("A")* should result in exactly the same receipt of published messages.

6) Length of subjects:

The documentation of the API does not specify a limit on the number of elements within a subject. Therefore, the number of elements within a subject should not matter. Considering a client system that subscribed to topic *"A"* by using *subscribe("A")*. Then, it should only receive messages with a subject *"A"*. According to our metamorphic rule, prepending multiple elements (e.g. *"B.C"*) to the subject string – resulting in *subscribe("B.C.A")* – should not result in any changes regarding the number of messages that are received when prepending exactly the same elements to the subjects of the messages to be published as well.

7) Length of elements:

Since there is not a strict limit that forbids subject elements having more than 12 characters, we define a rule such that the length of single elements should not matter. Similar to metamorphic rule 6) we change the subjects of messages on the subscription side along with changing the subjects of published messages in the same way.

True Equivalences for a given set of subjects

For testing statement sequences in the context of *True equivalence for a given set of subjects* we manually create statement sequences for the subscription side. In doing so, we explicitly consider the characteristics of the GMSEC API which we found out during our analysis of the API's functionality. Then, we automatically generate a set of subjects for messages to be published. When publishing messages with these generated subjects, a subscriber that executed all of the previously created statement sequences should receive all these messages. If this is not the case, the test case fails.

EXECUTION OF TEST CASES

Based on the procedure described in the previous sections we created 38 source test cases for the class of *True equivalences*. Based on the source test cases and the metamorphic rules we generated 374 follow-up test cases. Regarding the class of *True equivalences for a given set of subjects* we created a total of 37 test cases including 14 sets of one source and one follow-up test case and three sets with two follow-up test cases.

A test case consists of a set of subscription statements for a subscriber and a set of subjects for messages to be published by a publisher.

Once we have created a set of test cases, we create a folder structure as follows. Each source test case is placed in a folder which further contains the respective follow-up test cases along with a file which contains the generated subjects for the messages to be published. Then, we run our test program with the folders as input where the subscriber and publisher are implemented as independent threads. Please note, that the publisher only sends the messages as soon as the subscriber executed all subscription statements. A successful execution of a test case ends with sending and receiving a message with a subject that is defined a priori, for example "*LAST.MESSAGE*".

Results for test cases are stored in the folder that contains the respective test cases. In doing so, received messages are written in a separate text file. Subsequently, for each generated follow-up test case belonging to class *True equivalences* or class *True equivalences for a given set of subjects*, we automatically compare the subjects of the received messages with the received messages of the source test case to detect possible inconsistencies. The comparison is done by a bytewise comparison of the file contents. Thus, any deviation due to missing, extra, or changed order of messages would be detected.

FINDINGS

During our testing of the publish and subscribe functionality we found inconsistencies related to the functional behavior of the GMSEC API. Experimenting with our test program implementation (e.g. increasing the number of nullifying statements or changing the number of characters for a subject) we further discovered some issues that are related to non-functional behavior (e.g. the number of client systems that can be connected to the software bus). All of them might cause issues when using the GMSEC API and are described in the following sections.

No return object when excluding a subject

According to the documentation of the GMSEC API [7] the method that executes the subscriptions for given subject strings returns a *SubscriptionInfo* object after a successful execution. This object must be used to execute unsubscriptions from a subject. In this way it is ensured that unsubscriptions from subjects not subscribed to cannot occur. In contrast, the method that executes the exclusion of particular subjects does not return any object after its execution. Further, calling the method that removes an exclusion of a given subject does not give any feedback regarding whether or not the subject is successfully removed from the exclusion. Given this fact, two statement sequences of two independent, subscribing client systems which both receive exactly the same messages, can be manipulated such that they receive messages they did not received before. The problem is illustrated in the following. Consider two client systems that act independently from each other and earlier subscribed to the same subjects by executing the following equivalent statement sequences.

- System1:
  subscribe(".>")*
- System2:
  subscribe("+"),
  excludeSubject("")*

System1 subscribes to all subjects that contain at least two elements. The first element serves as a wildcard for elements, the ">" (greater than) extends the pattern by one or more elements, which together result in receiving messages whose subjects contain more than two elements. System2 first subscribes to all subjects that have zero, one or more elements. An exclusion is added to ensure that subjects that contain exactly one element are not received.
Since messages with empty subjects cannot be published, the case that System2 subscribes to subjects with zero elements can be neglected which results in the equivalence of the two statement sequences regarding the receipt of published messages. Now, System1 and System2 should receive exactly the same set of messages. Based on this example, we can exploit that the exclude method does not return any object that is related to the exclusion of a given subject and that it does not provide any feedback regarding the execution of the exclusion.
If we add the following nullifying statement sequence to the statement sequences of both client systems, the subscription might result in unintended receipt of messages. Further, the previous equivalence regarding the receipt of messages turns into a non-equivalence.

- *excludeSubject("*"),*
  removeExcludedSubject("")*

This results in the following statement sequences:

- System1:
  subscribe(".>"),*
  excludeSubject(""),*
  removeExcludedSubject("")*
- System2:
  subscribe("+"),
  excludeSubject(""),*
  excludeSubject(""),*
  removeExcludedSubject("")*

Since subjects with one element have been implicitly excluded in System1 by not subscribing to them, the explicit exclusion (*excludeSubject("*")*) is redundant. The subsequent removal of the exclusion of subject *"*"* results in no change of the previous receipt of messages. In contrast, adding the nullifying statement sequence to System2 results in changes regarding the receipt of messages.The *excludeSubject("*")* statement is a duplicate statement which can be executed without causing an exception or the system complaining (unlike to duplicate subscriptions for the same subject, for example). Subsequently, we remove the subject "*" from the exclusion. This results in a change since from now on messages that contain exactly one element in their subject are allowed again.
A subsequent execution of the test with the previously generated publish subjects confirms our assumption that the initial statement sequence equality turned into a statement sequence inequivalence after adding this nullifying statement sequence. To avoid potential future issues, we suggest that an object is returned by the exclusion method to provide a reliable removal of exclusions in the same way *unsubscribe* cannot be issued without a matching *subscribe*. Further, the documentation of the GMSEC API does not provide clarifying information about the functioning of either *excludeSubject(...)* or *removeExcludedSubject(...)*. For example, it is unclear whether the *excludeSubject(...)* method adds a duplicate subject to its internal storage structure or not. Also, the documentation for *removeExcludedSubject(...)* does not provide information whether exactly one occurrence of the subject to be removed from exclusion is removed or all occurrences (if duplicate subjects are allowed in the internal storage structure). Having this information would help developers to employ the above mentioned methods in a correct way and to avoid mistakes when using the GMSEC API.

Number of characters

We discovered that the subject for a *subscription* should not contain more than 65,534 characters; otherwise the program exits with a timeout error. This issue occurs independently of the number of elements and delimiters. In contrast, if a subject for the functions *excludeSubject* and *removeExcludedSubject* contains more than 65,534 characters the discovered timeout does not occur. Here, no feedback is provided by the GMSEC API such that no timeout error occurs. Instead, the testing program finishes without any error or notification. Since *subscribe* is not possible with that given number of characters, the unsubscription of this topic is also not possible as an unsubscription can only follow a subscription with exactly the same subject from which we want to unsubscribe.
Considering that mission critical software components use the GMSEC API, a timeout should be handled more explicitly to decrease the risk of a malfunctioning system.
After discovering this finding, we focused on the publishing side and tried to publish messages with more than 65,534 characters. In doing so, a subscriber subscribed to subjects that match the expression *"A.>"* and *"A.+"*. After sending some messages, it turned out that messages with a subject of more than 65,534 characters are dropped. All previous and subsequent messages which contain fewer characters and match the expression were received by the subscribing client system. Reflecting this behavior of the GMSEC API, at least an immediate feedback should be provided such that the publishing client system can react properly in the case of an unsuccessful publishing attempt instead of just dropping the message.

Publishing too many messages

Publishing too many messages results in an exception error on the publishing side thrown by the GMSEC API. We published 10,000,000 messages with randomly generated subjects whereby the subjects were limited to six elements and one character per element. Our test case logged the number of messages that had been published until the exception was thrown. Executing this test case several times, we discovered that this number depends on the system's resource utilization. For example, in one execution the exception was thrown after 3,323,963 published messages. While running another resource consuming application in parallel an execution was aborted after 2,409,800 published messages.

Too many clients connected

We assume, that this exit is related to the resource utilization of the system. Our testing software creates for each publishing and subscribing client system a separate thread so that we can run the different client systems in parallel. That means that for each new subscriber we also connect a new publisher to the software bus. After the subscribing system subscribed to all its subjects and the publisher published all its messages, we did not explicitly disconnect the client systems from the software bus. Instead, we just initiated an orderly shutdown of the created threads before we connect a new publishing and subscribing client system to the software bus. In doing so, only previously submitted tasks will be executed by the threads, new tasks will not be accepted. We tried to connect 302 client systems to the software bus, after the successful connection of 82 client systems the running application process was killed and exited with the exit code. When we disconnected the client systems properly after the execution of their tasks, i.e. using the appropriate cleanup method provided by the GMSEC API, our program exited properly without returning an error code. Nonetheless, this test revealed that too many client systems connected to the software bus will result in inappropriate results.

Given these issues found during our testing, we reported those issues to NASA.

RELATED WORK

For our testing approach on the functionality of GMSEC we used several techniques which are related to existing papers in the literature. These papers are described and discussed in the following against the background of our work.

Metamorphic Testing

An oracle problem exists if it becomes difficult or impossible to formulate the expected output. This problem is alleviated by metamorphic testing [3] which defines metamorphic relations that describe properties of the system under test given several inputs and outputs. An overview about different domains where metamorphic testing has been successfully applied is presented in the survey by Segura et al. [6]. In addition, they present different strategies for creating metamorphic relations, test cases and test execution strategies in the context of metamorphic testing.

In [11], the authors propose metamorphic relations for autonomous drone systems. Relations are modeled using extended finite state machine (EFSM) models and refer to geometric changes in the environment of the simulation, such as rotations of the map and different formations of obstacles. Using the models, numerous test cases are automatically generated where each should result in the same behavior [11]. Further, several measures are described to analyze the equivalence of test case results and possible deviations.

Segura et al. [5] present metamorphic relations in the context of feature models which represent valid combinations of features given a particular domain. Further, they present an automated test data generator for feature model analysis tools. Based on a manually created feature model, several variations of similar feature models are created by means of the proposed metamorphic relations. These models in turn serve as input for employed feature model analysis tools. Subsequently, the respective output of the analysis tool for each of the feature models is compared against each other.

Testing of Middleware-Based Systems

To the best of our knowledge, our work is the first that explicitly considers testing the publish and subscribe functionality of software busses in terms of a correct usage of subjects. However, there are other papers that consider testing the general usage of middleware-based systems, for example regarding the correct connection and disconnection of client systems to the software bus.

GMSEC's flexible software bus is tested in [16] by applying a model-based testing approach based on finite state machines (FSMs) as well as EFSMs. Abstract test cases are generated by means of FSMs and EFSMs and subsequently transformed into concrete, programming language specific test cases. Then, each of these test cases is applied on the GMSEC API considering three different middleware implementations which are used to transfer published messages. In contrast to their work, we only consider one middleware implementation, Bolt, which is one of the middleware implementations tested in their paper. In addition, we focus on the Java programming language. However, we explicitly test the behavior of different subscription statement combinations along with several automatically generated subject strings which is not considered in their paper.

An experience report of testing a software bus by means of MBT is given in [13]. Here, models are represented by means of process algebra specifications which serve as input for the JTorX tool [14] which is used for an automatic and online test case derivation. Test cases are run simultaneously during model exploration. Since we first generate test cases in an offline fashion before they are executed, our test cases are repeatable: In [13], test cases are derived and run simultaneously during the model exploration, and can change during runtime [15].

In [8], the authors present a checkpoint-based testing approach for integration testing of a context-sensitive middleware-based application, the so-called Reconfigurable Context-Sensitive Middleware (RCSM) [9] which makes use of a Situation-Aware Interface Definition Language (SA-IDL) [10] to define situation expressions. Situation expressions define which function is to be activated by the middleware given a particular external context change. If no function is activated for a certain amount of time, testers assume that the system is in a stable state where system-internal changes do not occur, and subsequently execute appropriate test cases. Such a stable state is seen as a checkpoint and marks the starting and ending point of a test case execution. The authors present a few example metamorphic relations which are applied to a smart delivery system. In comparison to this approach, we do not assume changes in the environment of the system under test during the execution of our test cases. In doing so, we assume without any check that there is a starting and ending checkpoint available at any time.

In the context of metamorphic testing revealing issues related to the non-functional behavior of a program, Segura et al. [22] describe a new idea of combining performance testing and metamorphic testing, the so-called *performance metamorphic testing*. In their vision paper the authors describe possible challenges of performance metamorphic testing by means of real-world examples. Further, possible metamorphic relations between performance measurements of the executions of the program to be tested are presented. Considering our metamorphic rules 6) and 7) we also found a way to apply metamorphic testing to find issues related to non-functional behavior.

Test Case Generation

In MBT [1] a testing model is used to derive the test cases and also serves as the test oracle for the test execution. In our work, we automatically generate subjects for messages to be published. Further, we automatically generate follow-up test cases using an algorithm for subject creations and following the metamorphic relations we defined.

Model-based metamorphic testing has already been successfully applied to NASA's Data Access Toolkit (DAT) [1]. Here, models were used to generate equivalent queries for both the DAT's REST and Web UI interface. Then, data sets returned by the executed queries are subjected to a pairwise comparison. This testing was conducted over several releases of DAT and resulted in a large number of detected defects that had not been reported before.

SUMMARY AND FUTURE WORK

We have presented a metamorphic approach for testing of pub-sub systems. The approach was applied to GMSEC, which is a mature software system that has been tested thoroughly over many years and is successfully used in many NASA missions. As a result, it has high quality and is a robust system. The approach was able to detect previously unknown issues of all which are corner cases that do not easily manifest themselves during regular testing. The reason not more issues were detected related to the fundamental pub-sub mechanisms is the rigorous testing and frequent use GMSEC has been exposed to.

Our metamorphic testing approach is conducted as black box testing, i.e. we have not analyzed the source code or used information about the source code to construct test cases. We have found that the presented testing approach shows potential for identifying corner cases where actual system behavior does not match expected behavior. When the corner cases have been addressed and all test cases pass, the metamorphic testing approach increases our confidence in the system. Based on our findings presented in this paper and the testing framework we have created, we plan to extend our testing activities of the pub-sub functionality of NASA's GMSEC middleware.

ACKNOWLEDGEMENTS

This work was supported by NSF grant 1446583.

REFERENCES

[1] M. Lindvall, D. Ganesan, R. Ardal, and R. E. Wiegand. Metamorphic model-based testing applied on NASA DAT: An experience report, in Proceedings of the 37th International Conference on Software Engineering - Volume 2, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 129–138. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819030

[2] D. Ganesan, M. Lindvall, L. Ruley, R. Wiegand, V. Ly, T. Tsui. Architectural Analysis of Systems Based on the Publisher-Subscriber Style. WCRE 2010: 173-182

[3] H. Liu, F. Kuo, D. Towey, and T. Y. Chen. "How effectively does metamorphic testing alleviate the oracle problem?" IEEE Trans. Software Eng., vol. 40, no. 1, pp. 4–22, 2014. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2013.46

[4] V. Gudmundsson, C. Schulze, D. Ganesan, M. Lindvall, and R. Wiegand. "An initial evaluation of model-based testing." in ISSRE (Supplemental Proceedings), 2013, pp. 13–14.

[5] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortes. Automated metamorphic testing on the analyses of feature models, Inf. Softw. Technol., vol. 53, no. 3, pp. 245–258, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2010.11.002

[6] S. Segura, G. Fraser, A. Sanchez, A. Ruiz-Cortes. A Survey on Metamorphic Testing, IEEE Transactions on software engineering, vol. 42, no. 9, September 2016

[7] GMSEC Interface Specification. Retrieved from https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20160005641.pdf, March 2016

[8] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, S. S. Yau. Integration testing of context-sensitive middleware-based applications: A metamorphic approach. International Journal of Software Engineering and Knowledge Engineering, vol. 16, no. 5, pp. 677-703, 2006

[9] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S.K. S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. IEEE Pervasive Computing, 1 (3): 33–40, 2002.

[10] S. S. Yau, D. Huang, H. Gong, and S. Seth. Development and runtime support for situation-aware application software in ubiquitous computing environments. In Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004), pages 452–457. IEEE Computer Society Press, Los Alamitos, California, 2004.

[11] M. Lindvall, A. Porter, G. Magnusson and C. Schulze. Metamorphic Model-Based Testing of Autonomous Systems, 2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET), Buenos Aires, 2017, pp. 35-41.

[12] A. Michlmayr, P. Fenkam, S. Dustdar. Specification-based unit testing of publish/subscribe applications. In: 26th IEEE international conference on distributed computing systems workshops 2006 (ICDCS workshops 2006)

[13] M. Sijtema, M. Stoelinga, A. Belinfante, L. Marinelli. Experiences with formal engineering: model-based specification, implementation and testing of a software bus at neopost. In: Formal methods for industrial critical systems

[14] A. Belifante. Jtorx: a tool for on-line model-driven test derivation and execution. In: Tools and algorithms for the construction and analysis of systems, 2010

[15] C. Schulze, M. Lindvall, S. Bjorgvinsson and R. Wiegand. Model generation to support model-based testing applied on the NASA DAT Web-application - An experience report, 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, MD, 2015, pp. 77-87.

[16] V. Gudmundsson, C. Schulze, D. Ganesan, M. Lindvall and R. Wiegand. Model-based testing of NASA's GMSEC, a reusable framework for ground system software, Innovations in Systems and Software Engineering, Springer, 2015, 11, 217-232

[17] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. ACM Computing Surveys, 35(2), 2003.

[18] https://gmsec.gsfc.nasa.gov/GMSEC%20FAQs%202014%2004%2023.htm#_Where_is/has_the, last accessed 2/8/2018

[19] T. Y. Chen, S. C. Cheung, S. M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998

[20] S. Segura, J. A. Parejo, J. Troya & A. Ruiz-Cortés. Metamorphic Testing of RESTful Web APIs. IEEE Transactions on Software Engineering, IEEE, 2017

[21] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. Tse, F.-C. Kuo and T. Y. Chen. Automated functional testing of online search services Software Testing. Verification and Reliability, Wiley Online Library, 2012, 22, pp. 221-243

[22] S. Segura, J. Troya, A. Durán & A. Ruiz-Cortés. Performance metamorphic testing: motivation and challenges. Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER), 2017 IEEE/ACM 39th International Conference on, 2017, pp. 7-10