2016 IEEE 27th International Symposium on Software Reliability Engineering

Experience Report: Model-based Test Automation of a Concurrent Flight Software Bus

Dharmalingam Ganesan[*], Mikael Lindvall[*], Stefan Hafsteinsson[*], Rance Cleaveland[^], Susanne L. Strege[+], Walter Moleski[+]

[*]Fraunhofer Center for Experimental Software Engineering, Maryland, USA, [^]University of Maryland, USA, [+]NASA Goddard Space Flight Center, Maryland, USA

{dganesan, mlindvall, shafsteinsson}@fc-md.umd.edu, rcleaveland@umd.edu, {susanne.l.strege, walter.f.moleski}@nasa.gov

ABSTRACT

Many systems make use of concurrent tasks, however it is often difficult to test concurrent design. Therefore, many test cases are simplified and do not fully test all concurrency aspects of the system. We encountered this problem when analyzing test cases for concurrent flight software at NASA. To address this problem, we developed and evaluated a model based testing (MBT) technique for testing of concurrent systems. Using MBT, the tester creates a model, which is based on the requirements of the system under test (SUT), and lets the computer generate innumerable test cases automatically from the model. We evaluate the effectiveness of the technique using Microsoft's Spec Explorer MBT tool. We apply the technique on NASA's Core Flight Software (cFS) software bus module API, which is based on a concurrent publisher-subscriber architecture style and is a safety-critical system. We describe how we created a test automation architecture for testing concurrent inter-task communication as carried out by the software bus. We also investigate the type of issues the technique for testing of concurrent systems can find as well as what degree of code coverage it can achieve.

Categories and Subject Descriptors
D.2.5 [**Testing and Debugging**]

General Terms
Testing, Safety, Quality, Test Automation.

Keywords Model Based Testing, Concurrency, Publish-Subscribe, Flight Software.

1 INTRODUCTION

Many systems make use of concurrent tasks, however it is often difficult to test such systems due to interleaving of execution of tasks. Therefore many test cases are simplified and do not fully test all concurrency aspects of the system. We encountered this problem when analyzing test cases for concurrent flight software at NASA. For example, NASA's Core Flight Software [3] (cFS) software bus (SB) module API is based on a concurrent publisher-subscriber architecture style [10]. In this flexible style, communication between modules is indirect, using the SB as the message broker. Modules concurrently use the SB to publish a message which will be delivered to all subscribers of the message. CFS is a safety-critical system. One problem is that safety-critical systems are expected to behave reliably also for off-nominal scenarios and thus significant testing is required in order to gain confidence in the system.

However, as mentioned above, fully testing cFS is non-trivial due to concurrency that cFS' software bus is based on. A SB is inherently difficult to test because publishers and subscribers may run on one or more tasks concurrently. To properly test a SB, the order in which publishers and subscribers interact with the bus needs to be controlled and coordinated, otherwise deciding the correctness of the bus is impossible. For example, if a publish event precedes all matching subscribe events, then the message shall be dropped and the subscribers shall not receive the message, which is a requirement in the cFS. On the other hand, if a subscribe event precedes the matching publish event, the message shall be delivered. Thus, the order of events is very important, and it is not enough to randomly generate publish and subscribe events using MBT to test the functional correctness of the bus without keeping detailed track of the state of the system at all times.

We addressed this problem by developing and evaluating a testing technique for testing of concurrent systems. The technique is based on Model Based Testing (MBT). The reasons for using MBT as the basis are grounded in the fact that systematic software test automation is necessary to avoid common testing problems. Software test automation is becoming increasingly popular with the introduction of emerging processes and tools, such as nightly builds, continuous integration, test-driven development, and test automation frameworks. Even though these approaches enable a high-level of automation through automatic execution of test cases and save cost as well as time, the very process of constructing the test cases is often manual. Manually written test cases are often "uneven," meaning that the test cases cover common usage scenarios but do not trigger so-called off-nominal or corner-case scenarios, which the tester has not thought of [1][2]. This limitation of manually written test cases is a significant problem for safety or mission critical systems, such as the NASA's reusable core flight software (cFS) [3] studied in this paper.

Another problem is that manually written test cases tend to have weak traceability to the requirements of the system under test (SUT) making such traceability links difficult to maintain during software evolution. In addition, test cases (i.e. test programs or scripts) are highly technical with the effect that the testing goal tends to disappear due to the large amounts of code fragments and programming constructs necessary to carry out the test. Altogether, this make test cases difficult for non-technical stakeholders to understand and it impedes determining whether or not all requirements are being tested. This is a problem especially for API-level testing (i.e. testing through the application program interface), because APIs are inherently technical, as discussed in this paper. In recent decades, MBT, a black-box testing technique, has gained great momentum [9] [11] [15]. In MBT, the tester creates a model based on the requirements of the SUT. The tester generates innumerable test cases automatically from the model. The generated test cases are executed against the SUT and the actual/observed behavior is automatically compared to the expected behavior encoded in the model, which in our work is created using Microsoft‘s Spec Explorer tool [14] [12].

We evaluate the effectiveness of our approach for testing of concurrent systems by applying it on NASA‘s cFS software bus (SB) module. The contributions of this paper is twofold. Our first contribution is that we create a test automation architecture for testing the inter-task communication as carried out by the SB. Our test architecture is inspired by a parent-child metaphor in which parents send commands to their children who perform the commands and send the result back to their parents to decide the overall correctness. We use our technique to generate different types of parents. Equivalently, each parent is a test case in that it sends a sequence of commands to its children, who are instantiated at runtime from a framework we developed. Our second contribution is an evaluation of the effectiveness of our MBT-based technique by addressing the following set of questions:

1. Is the technique, which is based on MBT, applicable to testing a concurrent publisher-subscriber system?
2. What type of requirement issues and functional errors can such a technique find?
3. What is the code coverage of such a technique?

The rest of the paper is structured as follows: In Section 2, the context of our work is presented. In Section 3, the test architecture and strategies are presented. In Section 4, modeling and test generation are presented. In Section 5, the strengths and weaknesses of MBT are discussed. In Section 6, related work is discussed.

2 CONTEXT

2.1 Missions using cFS

The interest in the cFS has been spreading fast within the aerospace community [3]. For example, the Lunar Reconnaissance Orbiter (LRO), the Global Precipitation Measurement (GPM), the Magnetospheric Multi-Scale (MMS) at NASA GFSC, the Radiation Belt Storm Probes (RBSP) at Johns Hopkins University/Applied Physics Laboratory (JHU/APL), and the Lunar Atmosphere and Dust Environment Explorer (LADEE) at NASA AMES use cFS, and many more missions are expected to use the cFS in the future. At the time of writing this paper, the cFS is being rated for manned missions. Thus, systematic and rigorous testing is crucial to uncover hidden bugs for human safety.

2.2 Architecture of the cFS

The cFS has a layered modular structure and is implemented in C [4] [3]. The top layer consists of mission independent modules called applications, which can be used in one or more missions. The second layer is the Core Flight Executive (cFE). The cFE is the core of the cFS and must be used in all missions. The third layer consist of the OS abstraction layer (OSAL), which offers a common API for all operating systems supported by the cFS (e.g. VxWorks, RTEMS and UNIX) [6].

2.3 Architecture of the Software Bus

The SB is the part of the cFS that is being tested. The applications are not allowed to communicate with each other directly via API calls. Communication between applications is instead conducted by passing messages through the SB, which is located in the cFS layer. That is, the applications communicate through message pipes by subscribing to and publishing messages from the SB. Figure 1 shows the context diagram of the SB in the cFS. Each application runs as a separate task, and communicates with other applications through the SB API. Each mission has three types of applications: 1) cFE core applications, which are required in each mission (the SB API is one of those applications); 2) optional cFS applications that missions can choose from; 3) mission specific applications.

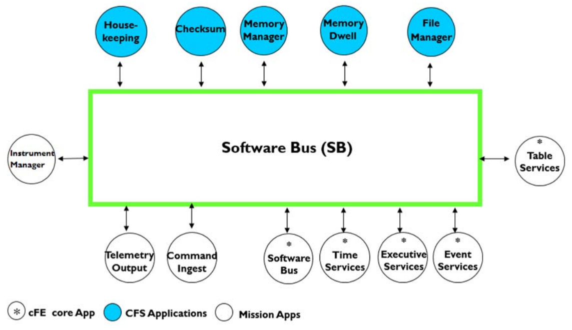

Figure 1: The context diagram of the software bus.

2.4 Testing Problem

The NASA team has already done a great job of developing a test suite of manual unit tests, reaching almost 100% line coverage [5]. However, the unit test suite does not take concurrency into account because it assumes that there is only one task. For example, the existing unit test cases assume that the publishers and subscribers are running on the same task, not testing the concurrent pub-sub features across task boundaries. To complement the existing test cases, we use MBT and generate test cases to test the SB where the publishers and subscribers can run on different tasks.

2.5 Collaboration Process

The NASA team develops the cFS software and does regular testing. The Fraunhofer team is an independent testing group that does testing after the NASA team has tested the system. The Fraunhofer team develops testing infrastructures, creates models for testing, generates test cases, runs test cases on UNIX, detects defects, reports defects to NASA, and delivers test cases for the NASA team to run in their lab. The Fraunhofer team visits the cFS lab to run the generated test cases of the SB on the flight software hardware and operating systems such as VxWorks and RTEMS.

3 THE TEST ARCHITECTURE

Since the CFS architecture dynamically loads tasks, the proposed testing architecture was deemed to be the best one for fully automatic testing. We chose the master slave metaphor since each app has no information about other apps' and therefore cannot determine whether a certain test case passed. Instead the master asks each slave to carry out commands before collecting results. This architecture tests concurrently running tasks in a controlled fashion and allows generating different types of masters that test different combinations of command sequences. In order to take concurrency into account when testing the MB, our testing is based on two complementary strategies.

Strategy 1: One parent app, multiple child tasks:
The objective of the first (basic) testing strategy is to test whether or not the SB properly routes the published messages across task boundaries but with a limited influence of concurrency. This is achieved by testing the SB API with only one parent app, which is generated using MBT. This parent app will create one or more child tasks. The child tasks will wait for commands from the parent app by pending on their command pipes, as shown in Figure 2. For example, if the parent asks child 1 to perform a subscribe command, then child 1 will call the subscribe function of the API and send the resulting return code to the parent. The parent will read the result from the result pipe (see Figure 2) and assert whether the return code matches the expected one as per the model. The result of the assertion is then logged in a log file for the tester to review. In this strategy, the parent will wait for the result from the child before sending a new command to other children. Thus, the parent coordinates the testing effort with its children. Even though the children are not concurrently running, this testing strategy is useful to test whether or not the SB properly routes the published messages across task boundaries, which is a missing feature of the existing unit test cases of the cFS. We leveraged the SB infrastructure for sending test commands from the parent to its children as well as for receiving results. It is a low risk to use the code of the SUT as part of the test infrastructure because we have embedded necessary assert statements in the source code of the child tasks that check that the behavior is correct. For example, if a child task creates a pipe to receive commands from its parent, the child will call the SUT's create pipe function with the valid arguments such as pipe name and depth. We assert that this should succeed because any valid task should be able to create a pipe. Therefore, there is no risk in using the SUT's functionality as part of the test infrastructure. All child tasks share the same code base, which are instantiated at runtime. Each child task subscribes to all of the SB API commands. The parent task sends out a sequence of commands based on the model of the SB. The SB broadcasts the commands. The child task with appropriate task id processes the incoming testing command and publishes the return code to the parent, again using the SB. If we get 50 test cases from our model, we view them as 50 parents, but we only run one parent at a time, otherwise we cannot reliably assert the actual output due to ordering of events.

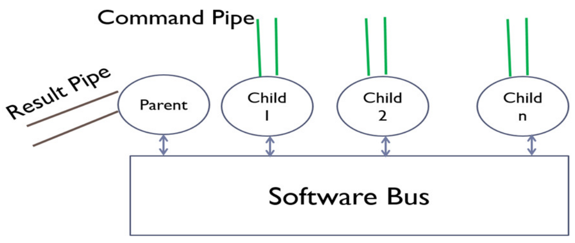

Figure 2: Architecture of strategy 1.

Strategy 2: Multiple parent apps, multiple child tasks: This testing strategy generalizes Strategy 1 by having multiple parent apps and thus a more complex and realistic concurrency situation is tested. Strategy 2 is more complex to set up, which is the reason we always start with strategy 1. All parent apps run concurrently, targeting concurrency related issues. Figure 3 illustrates the strategy. It shows, for example, 3 parent apps and each can spawn multiple child tasks. What these child tasks do is determined by its parent, which is based on the model. Since the tasks run concurrently, we had to make sure that the data parameters of each test case would not interfere with one another, otherwise our test cases will have a wrong test oracle, which is a fundamental problem in a publish-subscribe architecture.

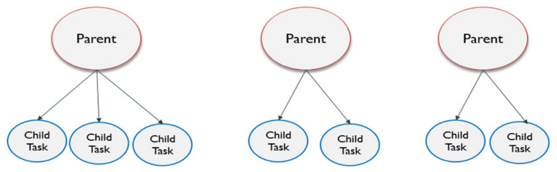

Figure 3: Architecture of strategy 2.

In our approach, we used the same model and ran the generated test cases using the above two different test strategies. This was done by automatically adjusting the data parameters (e.g. pipe ids, message ids) in the generated test cases using a template, discussed in Section 4.4. This adjustment ensures that each family of tasks will have their own set of message ids for subscribe, unsubscribe, or publish commands. If we have 50 generated test cases, we run only five test cases at a time, because it is difficult to reason about test assertion failures if there are too many concurrent tasks, although in production more than two dozen apps are plugged-into the bus.

4 Modeling and Testing of the Bus

In this section we give an overview of our testing environment and the workflow of how we used MBT on the SUT, the NASA's flight SB. It is worth to recall that we are viewing the SUT as a black-box, and that we are testing it through its API only. Figure 4 shows the architecture of the testing environment and the workflow. We used the Spec Explorer tool to develop model programs, which use a C# like language, and we use the tool to first automatically generate finite state machine models (FSMs) and then automatically generate test cases by traversing the states and transitions of the generated FSMs. We developed an adapter that converted the generated test cases into the SUT's native language, which is C. After running the test cases the tester examined the log to check the failed asserts.

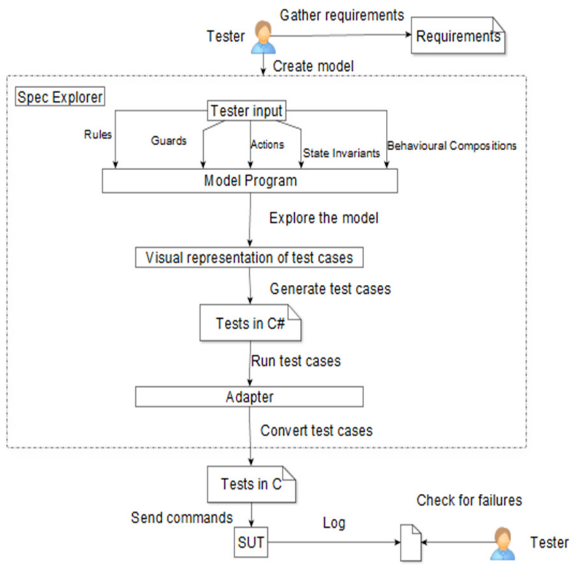

Figure 4: Architecture of the MBT infrastructure.

4.1 Model Creation

A model is typically created from a certain perspective of the SUT for a particular testing objective. Our objective is to test publish and subscribe features of the SUT, in the presence of multiple tasks, including off-nominal scenarios such as repeatedly subscribing and unsubscribing to a message. The model is a simplified, abstract representation of the SUT, because not all features of the SUT are modeled. It is an abstract representation because we reason about the SUT from a conceptual perspective, without dealing with the implementation details. It should be noted, however, that generated test cases are concrete and are executed against the full implementation of the SUT.

In Spec Explorer, MBT is performed by first developing a model program, based on the requirements of the SUT. Figure 5 shows the basic structure of a sample model program, which we will describe in more detail.

```
public class ModelProgram{
    +    State Variables
    +    Rule Methods
    +    Guards
    +    Domain Generators for Input Parameters
    +    Configuration Parameters
    +    State Filters
    +    Accepting State Condition
    +    State Invariants
}
```

Figure 5: Structure of a sample model program.

4.1.1 State variables

The state variables are data structures that keep track of which state the system is in, from the model's perspective. In our case, we need to keep track of the set of pipes that are created by each task, the set of messages each task subscribes to on a particular pipe, and the ordered list of messages received on each pipe. Figure 6 shows these state variables encapsulated in the `TaskData` structure. Since the goal is to test the SB in the presence of multiple tasks we have to track the `TaskData` of each task. `PipeDepths` tracks the maximum depth of each pipe, subscriptions tracks all the subscriptions on a pipe, and `msgPipes` tracks a list of message id's in a pipe that were received from other tasks. `TaskDataMap` contains the data for each task.

```
struct TaskData{
  // Key is the pipe name; Value is the pipe depth
  internal Map<int, int> pipeDepths;

  /* Key is the pipe name; Value is the list of
  subscriptions (list of msgIDs) */
    internal Map<int, Set<int>> subscriptions;

  /* Key is the pipe name; Value is the message pipe
    (list of msgIds in the pipe) */
    internal Map<int, Sequence<int>> msgPipes;

    internal Boolean missedMessages;
}

// Key is the task name and value is the task data
static Map<int, TaskData> TaskDataMap = new
Map<int, TaskData>();
```

Figure 6: State variables of the sample model program.

4.1.2 Rules and Guards

A rule is a method that updates some state variables. For every function to test, we have a rule method. Spec Explorer will automatically call the rule methods. Figure 7 shows an example of a rule where the SUT creates a task. The input parameter `taskName` is an integer that is supplied with the value using a so-called domain generator `TaskNameDomain`, which generates a set of parameter values. The rule has a guard that checks if the task can be initialized. If the guard is satisfied, the state variables are updated by the rule method, and the corresponding requirement cES1005 is captured for traceability. The result parameter is used as a parameter of the rule method, instead of a return code, because we let Spec Explorer choose parameter values automatically that satisfy the Boolean condition given in the `Condition.IsTrue` method. This allowed us to configure the rule method to test nominal cases (result = true) and off-nominal cases (result = false).

```
[Rule]
public static void
InitTask([Domain("TaskNameDomain")] int taskName,
bool result){
    bool guardStatus = CanInitTask(taskName);
    Condition.IsTrue(guardStatus == result);

    //Update the state variables.
    AddTask(taskName);
    if(guardStatus) Requirement.Capture("cES1005");
}
```

Figure 7: A rule method fragment to create a task.

Figure 8 shows an example of a guard that is used in the previous rule. This guard makes sure that a task can only be created if it is not already created, by checking that the `TaskDataMap` doesn't already contains the `taskName`.

```
public static bool CanInitTask(int taskName){
    return !TaskDataMap.ContainsKey(taskName);
}
```

Figure 8: The guard that checks if a task exists.

4.1.3 *State filters*

A *State filter* excludes states that violate a condition. Figure 9 shows an example of a state filter, which excludes states that violate the bounds of the model's configuration parameters, such as the maximum number of tasks, pipe depth, etc. In our model program, we also configure the state filters so that we can generate off-nominal tests that violate the boundary conditions of configuration parameters and check whether the SUT actually fails or not.

```
[StateFilter]
static bool StateFilterRule{
    get{
        return !HasMaxPipeDepthSize() &&
        TaskDataMap.Count <= Constants.MAX_APP;
    }
}
```

Figure 9: State filter that excludes the unwanted states.

4.1.4 *Accepting state conditions*

Accepting states serve as a final state where each test case can successfully end. This means that at the end of the test case the system will be in a stable state for further testing. *Accepting state conditions* guide the model exploration algorithm to reach a final state with a certain property that applies to each test case. Figure 10 shows an accepting state condition where all tasks are deleted.

```
[AcceptingStateCondition]
static bool AcceptingState(){
     return TaskDataMap.Count == 0;
}
```

Figure 10: Forcing all test cases to delete all child tasks.

4.1.5 *Model verification using invariants*

The model program needs to be verified for modeling errors. The verification is done with the help of state invariants, which must hold true at every state during the model analysis phase. Figure 11 shows an example of a state invariant that the model program can always delete a task it has created, which is requirement cES1006.1 in the requirement specification. By creating the state invariants for each of the requirements, we were able to formally verify that the model program is correct.

```
// If a task is initialized, it can always be deleted
[StateInvariant]
public static bool InitDeleteTask(){
    return !TaskDataMap.Exists(t =>
    !CanDeleteTask(t.Key));
}
```

Figure 11: State invariant example.

If we change our rule methods to allow deletion of non-existing tasks, then we would violate the state invariant in Figure 11, resulting in a modeling error. Figure 12 shows that a modeling error has occurred in state S7.

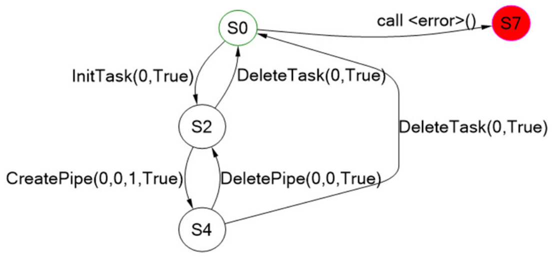

Figure 12: An invariant violation example.

4.1.6 *Model exploration to generate FSMs*

The goal of the model exploration step is to generate an FSM from the model program. It is often the case that the default exploration will run out of memory because, for example, the parameters of rule methods are often unbounded (e.g. int types). The tester must make the model program's state space finite by using the constructs of the Spec Explorer tool, which provides a scripting language called *Cord* to define so-called *machines* to make the exploration finite. Machines can be viewed as finite slices of an infinite state space. To obtain a visual representation of the model program as an FSM, the tester has to configure the model program using the Cord scripting language. There are two types of machines the tester has to create, a machine for exploring the model and one to generate test cases, see Section 4.2.

4.2 Layered testing using scenarios

Scenarios are used to limit the behavior of the model program to a selected subset of features to be tested. Our strategy is always to first test the most basic features before testing more advanced features and combinations of features. We achieve this by slicing the model program in different ways. This strategy helps in making the generated test cases easier to understand and debug. Figure 13 shows an example of a scenario to only test create (or delete) tasks and pipes. The underscore symbol (_) indicates that we want Spec Explorer to supply our rule methods with appropriate parameters in such a way that the result parameter will be true. In this testing scenario, the tester wants to only test nominal SUT usage, that's why the return code of the all functions are all restricted to `true`. Off-nominal scenarios are tested by replacing `true` with the underscore symbol (_), which will cover `false` return code. The composition operator `|||` is used here to compose multiple actions, which is also used for composing different machines later.

```
machine PipesScenario() {
  InitTask(_,true)* ||| DeleteTask(_,true)* |||
  CreatePipe(_,_,_,true)* |||
DeletePipe(_,_,true)*;
}
```

Figure 13: An example scenario specific machine.

Figure 14 shows a machine for exploration of the sliced model. It is a combination of the scenario in Figure 13 and the default model program machine called `ModelProgram`.

```
machine PipesProgram() {
   PipesScenario || ModelProgram
}
```

Figure 14: An example machine to generate an FSM.

To generate a visual FSM, the tester explores a machine. Spec Explorer will try to apply every rule, apply filters to remove unwanted states and verify model invariants. Figure 15 shows an example of a generated FSM based on the machine in Figure 14, where the green circle is the accepting state in which all tasks are deleted. The intermediate states S1 and S3 are not shown because we collapsed the call and return actions into one action, a visualization option of the tool.

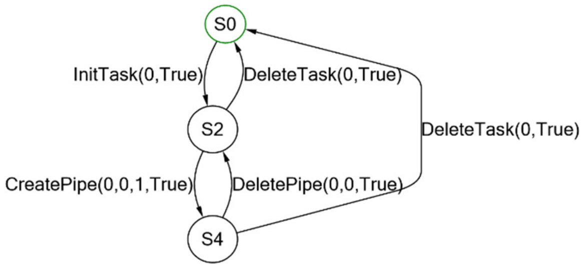

Figure 15: Generated FSM for Figure 14.

4.2.1 *Generating the test cases*

In order to generate test cases, the tester first creates a machine for test generation. Spec Explorer can construct test cases with two strategies, namely short tests and long tests. For both strategies, a full path coverage of the FSM is guaranteed, because both strategies contain at least one test case where each step or transition is taken. Short tests are basically the shortest paths taken through the FSM that ends in an accepting state. The generated short test cases are easy to read and debug when a test case fails. On the other hand, the generated long tests are generated from a long path taken in the model that ends in an accepting state. The exploration takes as many steps as it can, looping around transitions, attempting to trigger off-nominal scenarios, and ending in an accepting state. With this strategy, fewer and longer test cases are generated. If a tester wants to run test cases that could run for hours and test the system more comprehensively, then long tests would be the better option. Figure 16 shows a machine for generating short tests. By exploring this machine, the tester gets a visual representation of each test case, see Figure 17. The grey state is the starting state and the green circle is the accepting state. Similarly, long test sequences can be generated by setting the strategy to `longtests`.

```
machine Pipes() {
    construct test cases where strategy =
"shorttests" for PipesProgram()
}
```

Figure 16: Machine to generate test cases of Figure 15.

Spec Explorer can also generate test code for each test case chain in Figure 17. The generated test cases are in C#, which is the language Spec Explorer supports.

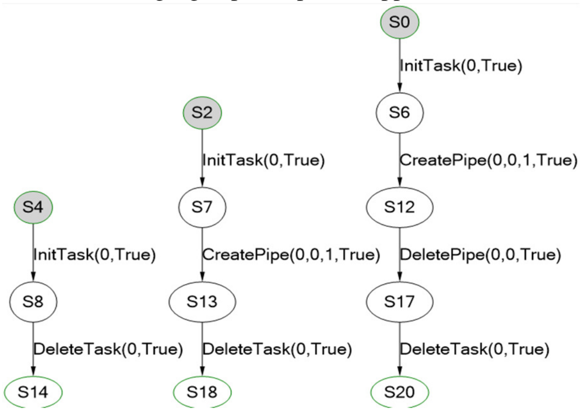

Figure 17: Generated test cases from Figure 14.

4.2.2 *Composition of scenarios for testing*

Once the basic scenarios are tested, the tester composes the already tested scenarios to test for new and complex combinations of off-nominal scenarios. Figure 18 shows how the `SubUnsubScenario` is composed with the existing `PipeScenario` from Figure 13. Exploration of this new machine will result in the FSM in Figure 19.

```
machine SubUnsubScenario() {
 UnSubscribe(_,_,_,true)* ||| Subscribe(_,_,_,true)*;
}

machine PipesProgramWithSubUnsub() {
 (PipesScenario ||| SubUnsubScenario) || ModelProgram
}
```

Figure 18: Composition of two scenarios.

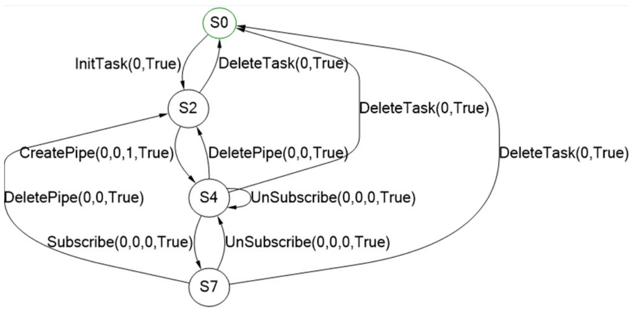

Figure 19: Generated FSM of the composed scenarios.

4.3 Capturing the requirements

Spec Explorer allows a tester to capture requirements in the model program. For example, Figure 20 shows the FSM where the requirement ids are captured for each transition based on the SUT's requirement specification.

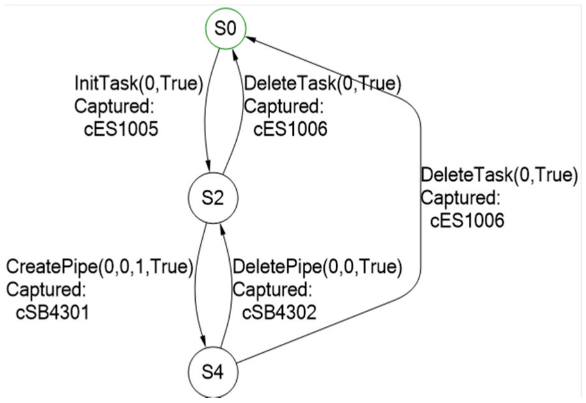

Figure 20: Generated FSM with captured requirements.

4.3.1 Slicing for requirements coverage

The tester can also slice the model program to cover a set of requirements of interest. Figure 21 shows a machine that slices the `PipesProgram` machine for requirements using the selective strategy of Spec Explorer. It is also possible to slice the model program by a given requirement id, too.

```
Machine PipeWithRequirementsProgram() {
    construct requirement coverage
    where strategy = "selective" for PipesProgram()
}
```

Figure 21: Slicing for requirements coverage.

When the machine is constructed using the selective strategy, a step is only create once for every requirement that is captured. Figure 22 shows the resulting FSM from exploring the machine. Since this FSM is constructed with the selective strategy, the transition S4 -> S0 in Figure 20 is eliminated. Figure 23 shows the generated test cases, with requirement ids for each step, for the FSM in Figure 22.

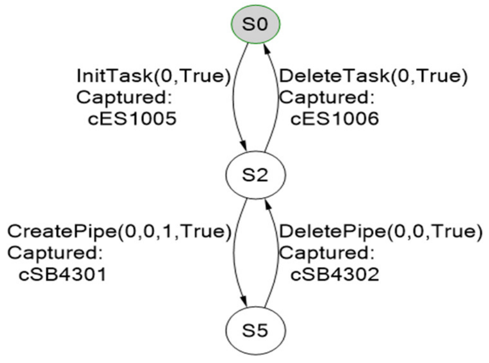

Figure 22: Generated FSM for Figure 21.

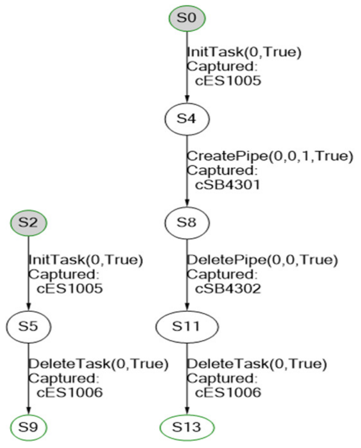

Figure 23: Generated test cases with requirement ids.

4.4 The Adapter

At this this point, our test cases are in C# and also tightly coupled to the Visual Studio environment. The adapter bridges the C# test cases generated to the SUT's C language.

4.4.1 Converting test cases in C# to C

To convert the generated test cases to the C language, we have defined templates which define the code structure of the generated test cases. The template allows the tester to manipulate the structure of the test cases easily. Figure 24 shows a basic structure of a template. For example, the `TEST_SEQUENCE` is a placeholder for the adapter to replace the generated test case in C. The `CreatePipe` and `AssertResults` labels are example actions that are parameterized and will be replaced by the corresponding concrete values determined by the model program.

```
Void <testCaseName>_Main( void ) {
        int32 Status = 0;
        <TEST_SEQUENCE>
}
#begin CreatePipe
    Status = CreatePipe_w(TASK_<taskName>,
PIPE_<pipeName>, <pipeDepth>);
#end CreatePipe

#begin AssertResult
        ASSERT_TEST(Status <expectedResult>,
Status, "<testCaseName>");
#end AssertResult
```

Figure 24: Template fragment for converting C# to C.

For example, consider the transition S4 -> S8 in Figure 23 and examine how the action `CreatePipe(0,0,1,True)` will be converted into C code. The adapter will exchange the first three numbers for the placeholders in the `CreatePipe` label, first the task name, second pipe name and last the pipe depth. The `expectedResult` placeholder in the `AssertResults` label will be exchanged with the `Boolean` value in the action based on the model program. One benefit is that we were able to change the template and generate test cases to fit our two testing strategies that were discussed in Section 3, without making any changes to the model. For example, we can change the message ids to avoid interference when multiple parent tasks are running as in test strategy 2.

4.4.2 *Traceability*

When the test cases were generated, the tester automatically gathered data on which requirements were covered by a test case, see Table 1. MBT helps maintaining traceability links between test cases and requirements, which is important for safety-critical systems.

Table 1: Generated traceability matrix (fragment)

Test suite	Requirement covered
CreateDeletePipeTestSuite	cES1005
CreateDeletePipeTestSuite	cES1006
SubUnsubTestSuite	cSB4303
SendRcvMsgTestSuite	cSB4305
SendRcvMsgTestSuite	cSB4308

4.4.3 *Testing config parameter bounds*

The model program has configuration parameters that bound the size of the generated model (e.g. Max number of pipes, Max number of apps). The SUT also has configuration parameters, defined in the C header files. Thus, the SUT's configuration parameters must be consistent with the model program because we want to make sure that the test assertions are consistent with the bounds of the configuration parameters. E.g., when the model creates a pipe of depth that exceeds the configured limit, that off-nominal input should result the SUT to fail. We want ensure that the SUT will behave properly by using the same configuration as in the model. We generate a header file for every test suite using the configuration of the model program.

5 ANALYSIS

5.1 Code Coverage of the SUT

MBT is a black-box testing technique, therefore it may not cover all code statements. It may, e.g. be impossible to cover behaviors that are undocumented as requirements because they were not known to the tester during modeling. We measured the coverage using gcov. To be fair to MBT, we only measured the code coverage for the API functions that were part of the model program (i.e. only code belonging to the SB functionality that is reachable directly or indirectly by calling the API functions), so the coverage discussed here does not apply to the whole SUT, although we tested and found defects in other modules that are transitively used by the SB module. The line coverage of the functions that were modeled is displayed in Table 2, excluding the source code comments. Note that the init and delete task functions are part of a different module that the SB uses. But in order to test the SB we have to initialize some tasks and delete them afterwards. Thus, we modeled these two "helper" functions.

The `SendMsg` function, which publishes a message, has the lowest code coverage because our basic model program was not handling the off-nominal scenario of sending malformed messages, which is not that explicit in the requirement document. Thus, a large block of error handling code was not covered. Similarly, we investigated the source code of the subscribe function and found that there are a few hidden requirements that were not modeled. For example, there is a hidden requirement that limits the maximum number of messages a subscriber can subscribe to, which was not included in the model. Although the source code size is small, the testing challenge is non-trivial due to concurrency.

Table 2: Code coverage of the SUT

Function	Lines hit	Coverage
`InitTask`	26 of 40	65%
`DeleteTask`	21 of 29	72.4%
`CreatePipe`	48 of 51	94.1%
`DeletePipe`	48 of 54	88.9%
`Subscribe`	63 of 82	76.8%
`Unsubscribe`	49 of 61	80.3%
`SendMsg`	71 of 130	54.6%
`ReceiveMsg`	35 of 35	100.0%
Total	361 of 482	74.9%

5.2 Types of Issues Found

This subsection provides answers to the questions in the introduction by explaining the different types of issues that were detected by MBT, see Table 3.

Table 3: Types of issues found

Issue Type	Number of issues
Duplicate requirements	1
Unspecified requirements	12
Issues in the test infrastructure	4
Functional issues in the SUT	5

5.2.1 Requirements Issues

Issue 1: *One pair of duplicate requirements was detected.* This issue was detected when the tester tried to, for traceability purposes, add the requirement number to the model program. When searching for the requirement, the tester found two requirements with two different numbers for this one behavior. This issue is already removed in the current version of the requirements document.

Issue 2: *Twelve unspecified requirements were detected*. The tester, when creating the model program, realized that a certain transition was necessary but couldn't find a requirement for it, so therefore captured such requirements as unspecified. Figure 25 shows an FSM that has missing requirements for some transitions taken by the model program. The missing requirements were captured with the keyword "Unspecified", indicating they were not in the requirement document. Most of the unspecified requirements are off-nominal, for example, how the SUT should react to deleting a pipe which was already deleted is not discussed in the requirements document, although handled by the API with appropriate failure return codes.

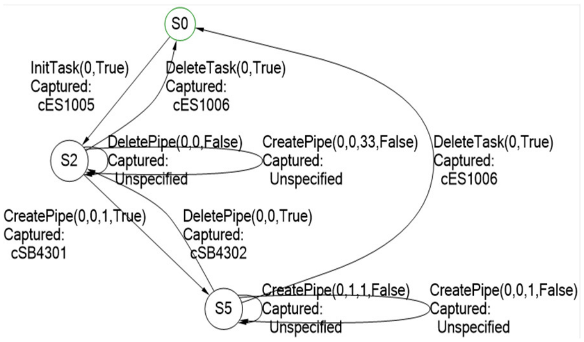

Figure 25: FSM with unspecified requirements.

5.2.2 Issues in the test infrastructure

The test infrastructure includes the model program, machines, adapter in Visual Studio, and the C wrappers of the SUT. It is worth noting that the generated test cases identified issues in the test infrastructure. Such issues can be viewed as false-positives from the SUT standpoint, but from an MBT point of view they are true-positives.

Issue 1: *Our wrapper was not translating some of the test cases correctly.* This issue was detected when we were deleting pipes that were not created. In our wrapper we were keeping track of which pipes existed and if a pipe did not exist, the container that was keeping track of the pipes would return pipe name as "0" for that pipe. In the SUT the pipe with the name "0" is owned by other tasks of the SUT, so rather than trying to delete a non-existing pipe, the wrapper told the child task to delete the pipe of some other tasks. Thus, we were not getting all expected coverage.

Issue 2: *Thread race condition.* We were using a logger that was protected by semaphores when an action wanted to log. When we were deleting a task there was a race condition between logging and deleting the task, sometimes resulting in task deletion but not giving up the semaphore.

Issues 3 and 4: *Duplicate subscribe or unsubscribe fails.* These two issues turned out to be a modeling error because the tester misunderstood two of the requirements. The tester assumed that subscribing to a message which was already subscribed to, should return false, but the test cases failed because SUT returns true as per the requirements. Similarly, unsubscribing an already unsubscribed message returns true but our model program wrongly assumed that it should be false. We fixed the model and regenerated a test suite.

5.2.3 Functional Issues in the SUT

All functional issues in the SUT were detected by the generated test cases using MBT. All functional issues were reviewed by NASA team, confirmed to be previously unknown and included into the discrepancies report for further actions. Most of the detected issues are off-nominal.

Issue 1: *Child task was not deleting all of its resources upon exit.* The child task should delete resources such as semaphores and message pipes when the parent sends a cleanup command, but in one case cleanup did not occur.

Issue 2: *Infinite loop when a child task exits.* The SUT incorrectly stayed in an endless loop until being terminated from the keyboard. We found that a test case sequence corrupted the internal state variables of the SUT.

Issue 3: *A pipe could receive more messages than allowed.* This issue was found when a test case created a pipe with pipe depth one, only allowing the pipe to store one message at a time. When the pipe subscribed to a message and that particular message was sent twice by a publisher, the pipe could read the message twice even though its pipe depth was only one. This issue has been resolved by the cFS team.

Issue 4: *Creating the same message pipe twice will not allow it to be deleted.* This issue was found when a test case sent the create pipe command (with the same data parameters) twice followed by the delete pipe command. This off-nominal test case has failed because the internal variable of the SUT that stores the generated pipe name after the first call was corrupted when called twice in a row.

Issue 5: *Dynamic loading and unloading of modules with the same entry point function name has failed*. This issue is caused by the OS abstraction layer (OSAL) of the cFS, which we used to run the test cases. We compiled each parent into a shared object and dynamically loaded using OSAL functions. However, all our parent code had the same entry point function name. This caused a problem in that all but the first shared object were never executed. Thus, we were running only the very first test case because the dynamic unloading feature did not work as expected. To overcome this issue, we generated a unique entry function point name for each of the test case and ran all test cases.

5.3 Scalability

If the model program is not carefully configured, the default exploration for generating an FSM crash the tool after generating 76,534 states and 75,926 transitions, which happened in our case. A model program with just 10 rule methods with each method taking 3 arguments (even as a Boolean type) will result in a state space of several million states. This is a severe problem when testing for off-nominal scenarios such as deleting an already deleted pipe or incorporating all combinations of off-nominal parameter inputs too. Thus, to scale MBT we had to apply some modeling tactics using scenarios and abstractions. 1) We sliced the default infinite state space by defining the scenarios of interest, starting from the most basic features to test. Scenarios helped in reducing the state space because we added constraints on rule methods and their parameter combinations to test. 2) We reduced the size of the state space by defining an abstraction on the message pipe. E.g., abstraction of the message pipe by only counting the number of messages, instead of keeping track of the actual messages, reduced the size of the state space to half because permutations of message ordering is abstracted way. This abstraction is useful to test whether the SUT delivers a message, but not to test the order of delivered messages. This is an example of a modeling trade-off to scale MBT.

6 RELATED WORK

In our own work, we have evaluated model-based testing on different types of real-world systems [13] [1] [7] [2]. This paper contributes our experience of testing a flight SB. In [13], we tested NASA's ground system. We addressed the testing challenges when multiple tasks are running concurrently in contrast to [13]. The modeling paradigm in this paper allows for generation of relatively large state spaces in contrast to the FSMs that were designed manually in [13]. Sijtema et al. have used MBT to test a SB, developed at Neopost Inc [8]. The bugs they reported are similar to our bugs. Their models are based on a powerful formal language called mCRL2, in contrast to our models which are programmed in a C# like language, making our models easier to construct for testers who are not trained in formal methods. We used code coverage to identify behaviors that were missed in the model, and to update the model to cover missing behaviors for achieving better coverage. Kicillof et al. [16] combined model-based testing for test procedure generation with symbolic execution (at code level) for data parameter generation in order to achieve code coverage for .NET applications. Modex tool [17], which extracts models from C code for model checking, provides stronger confidence than our testing, however the user needs to know the implementation details of the SUT.

7 Conclusions

Safety-critical systems must be extensively tested, not only for compliance to requirements, but also for behaving reliably for off-nominal scenarios. MBT has shown promising results in addressing these issues. However, concurrent systems cannot be tested using MBT without modifications. In this paper, we evaluated the effectiveness of our MBT-based technique by applying it on NASA‘s cFS software bus module using the Microsoft‘s Spec Explorer tool. We described our test automation architecture for testing the inter-task communication. We showed that it is feasible to apply to a SB such as the one CFS uses and that the technique can detect four different types of issues.

Acknowledgments

This work is partly supported by NASA's Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) and NSF CPS award "1446675". We thank Martha Wetherholt, Markland Benson, Steven Hard, and Kenneth Rehm for their support. Asa Bjork Valdimarsdottir helped in constructing the test infrastructure.

8 References

[1]. C. Schulze, D. Ganesan, M. Lindvall, D. McComas, and A. Cudmore, "Experience Report: Model-based Testing of NASA's OSAL API," in *ISSRE*, 2013.

[2]. C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, D. Goldman, "Assessing model-based testing: an empirical study in industry." *In Proc. ICSE '14 – SEIP*, 135-144.

[3]. Core Flight System, https://cFS.gsfc.nasa.gov/

[4]. D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew, "Verifying architectural design rules of the flight software product line." *In Proceedings of the 13th International Software Product Line Conference (SPLC '09)*. IEEE Computer Society Press, 161-170.

[5]. D. Ganesan, M. Lindvall, D. Mccomas, M. Bartholomew, S. Slegel, B. Medina, R. Krikhaar, C. Verhoef, L. Montgomery, "An analysis of unit tests of a flight software product line." *Sci. Comput. Program.* 78, 12, 2360-2380, 2013.

[6]. H. Femmer, D. Ganesan, M. Lindvall, and D. Mccomas, "Detecting Inconsistencies in Wrappers - A Case Study." *In Proceedings of the ICSE '13 – SEIP*, 2013.

[7]. M. Lindvall, D. Ganesan, R. Árdal, and R. Wiegand, "Metamorphic model-based testing applied on NASA DAT: an experience report." *In Proceedings of the 37th ICSE -* SEIP, Vol. 2, 129-138.

[8]. M. Sijtema, A. Belinfante, M. I. A. Stoelinga, and L. Marinelli, "Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at Neopost." *Sci. Comput. Program.* 80, 188-209.

[9]. M. Utting and B. Legeard, "Practical Model-Based Testing: A Tools Approach," Kaufmann Publishers Inc., 2007.

[10]. P. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe." *ACM Comput. Surv.* 35, 2 (June 2003), 114-131.

[11]. R. V. Binder, B. Legeard, and A. Kramer, "Model-based Testing: Where Does It Stand?," *ACM Queue* 13, 1.

[12]. Spec Explorer, www.microsoft.com.

[13]. V. Gudmundsson, C. Schulze, D. Ganesan, M. Lindvall, and R. Wiegand, "An Initial Evaluation of Model-Based Testing," in *IEEE 24th ISSRE*, 2013.

[14]. W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation : tools and methodology," *Softw. Testing, Verif. Reliab.*, vol. 21, no. 1, pp. 55–71, 2011.

[15]. H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Trans. Softw. Eng. Methodol*. 22, 1, Article 6 (March 2013).

[16]. N. Kicillof, W. Grieskamp, N. Tillmann, and V. Braberman, "Achieving both model and code coverage with automated gray-box testing," *Proceedings of the 3rd international workshop on Advances in model-based testing* - A-MOST '07, pp. 1–11, 2007.

[17]. G.J. Holzmann and M.H. Smith, "Software Model Checking: Extracting verification models from source code Software," *Testing Verification and Reliability*, 11 (2), pp. 65-79, 2001.