2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Metamorphic Model-based Testing Applied on NASA DAT –an experience report

Mikael Lindvall*, Dharmalingam Ganesan*, Ragnar Árdal*‡, and Robert E. Wiegand†
*Fraunhofer USA Center for Experimental Software Engineering (CESE), MD, USA
‡School of Computer Science, Reykjavík University Reykjavík, Iceland
†NASA Goddard Space Flight Center, Maryland, USA

***Abstract*—Testing is necessary for all types of systems, but becomes difficult when the tester cannot easily determine whether the system delivers the correct result or not. NASA's Data Access Toolkit allows NASA analysts to query a large database of telemetry data. Since the user is unfamiliar with the data and several data transformations can occur, it is impossible to determine whether the system behaves correctly or not in full scale production situations. Small scale testing was already conducted manually by other teams and unit testing was conducted on individual functions. However, there was still a need for full scale acceptance testing on a broad scale.**

We describe how we addressed this testing problem by applying the idea of metamorphic testing [1]. Specifically, we base it on equivalence of queries and by using the system itself for testing. The approach is implemented using a model-based testing approach in combination with a test data generation and test case outcome analysis strategy. We also discuss some of the issues that were detected using this approach.

I. Introduction

Testing typically involves comparing the actual outcome of a test case to the expected one. The mechanism to determine if the test cases has passed or failed is called the oracle [1]. If the oracle determines that the test case fails then the reason for the difference is investigated. There are many types of systems where the use of an oracle is applicable. For example, systems where the initial state is well-known and the system clearly indicates what the new state is, based upon a certain stimuli or request. For example, a medical device such as an infusion pump has these properties. The pump is initially off and all alarms are cleared. When a sensor detects a dangerous change in the blood pressure, then an alarm should go off. If the blood pressure goes back to normal within a certain time limit, then the alarm should be cleared, etc. For such systems we can predict the outcome and compare it to the actual. For example, we can test whether the infusion pump indeed shuts off within the specified time if the blood pressure stays abnormal. However, there are other types of systems and situations for which it is difficult to exactly predict the outcome for a given scenario. Examples of such systems are climate models that predict the climate 80 years into the future [2]; and Google maps that calculate directions from one arbitrary point on the map to another, and the distance between them [3]. For these systems, we have a test oracle problem because it is difficult to determine the expected output and to determine whether the actual outputs agree with the expected outputs [1].

This paper describes how we are addressing acceptance testing of NASA's Data Access Toolkit (DAT) system, which is still under development. DAT consists of a huge database of telemetry data and a powerful query interface with many degrees of freedom. Large amounts of data about various NASA missions and their individual instruments are collected and coded in a systematic way using various data collection systems. These large sets of data are then ingested into DAT. The last DAT database we received from NASA contained 26 GB of telemetry data. NASA scientists and engineers will use DAT to analyze, for example, the behavior of the same type of instrument across many different missions in order to detect various behavior patterns such as situation-dependent instrument failures. The users will typically formulate advanced queries that include and exclude data points based on mission, instrument (mnemonic), time intervals, sampling criteria and aggregations (when the resulting data set is expected to be too large), etc. The resulting data set will then be analyzed and visualized using built in DAT features, but can also be exported to external tools for more advanced analysis. Since the user is unfamiliar with the large body of underlying data being queried and since the data is transformed (e.g., due to sampling), it is impossible for the user to determine whether the resulting data set is absolutely correct or not. However, analysis of this type of spacecraft data must be correct and accurate. Thus, the user needs to be assured through testing that no data points are incorrectly excluded or included, that sampling and aggregation functions deliver the correct results, etc.

In addition, the DAT system is being developed using agile principles, which means that new releases will be delivered on a relatively frequent basis [4][5]. Each new release includes new and changed software functionality, new and changed database schema, as well as a new set of telemetry data. Thus, in order to facilitate rapid testing of a new version, the testing infrastructure also needed to be as agnostic to the underlying implementation, database schema, and data set as possible. Since the query language is also subject to change, the testing infrastructure also needed to be able to quickly update the testing environment and generate a large number of test cases and automatically execute and analyze those test cases and report issues to the NASA team within 24 hours.

In this paper, we describe how we addressed this testing challenge where we have an oracle problem.

The basic idea for our approach is based on the principle of metamorphic testing [1]. Specifically, we base it on equivalence of queries and by using the system itself for testing. This is done both by formulating the same query in different ways and by translating the query into different but equivalent query languages and then asserting that the resulting data sets are the same.

The approach is implemented using a model-based testing approach in combination with a test data generation and test case outcome analysis strategy. The model captures the grammar of the query language and allows us to generate a large number of different abstract test cases (i.e., high level description of a test case that lack details that would make it executable). Thus, all information is stored and maintained in the model, allowing for rapid change. The test data generation strategy analyzes the underlying telemetry data and determines critical time intervals in order to generate a large number of concrete test cases (i.e., a test case where the necessary details for the test case to be executable have been added) for each abstract test case. Thus, the test case generation is independent of the underlying telemetry data, which allows for rapid change. The test case outcome analysis strategy generates a new query, independent of the DAT query. The independent query is used to query an independent copy of the underlying telemetry data. The resulting data from the independent query should match the resulting data set from the DAT query. Thus, an oracle (the expected result set) is being automatically generated without knowing anything about the underlying telemetry data. This allows for rapid change since new test cases can be automatically generated once a new data set has been delivered. In addition, the entire testing infrastructure is installed on a separate virtual machine, in parallel to the virtual machine that DAT is installed on. Thus, when a new version of DAT is delivered, that virtual machine is replaced and up and running within minutes. This architecture adds to the ability to rapidly install and test a new version of DAT.

II. The Data Access Toolkit (DAT) System

DAT, which is the system under test (SUT), is a multi-satellite data archive, access, trending, and analysis software system that can support a variety of missions at NASA Goddard Space Flight Center (GSFC). Using standard data definitions, DAT ingests stored data to create a data archive comprised of original, processed and statistically reduced data for multiple satellites [6].

Once data is archived into the DAT repository, DAT provides an advanced query interface for mission analysts, mission managers, and the flight operation team to search and mine the available data.

For data access and analysis, an open interface is provided to the archive so that users of existing, custom, and commercial components can easily access the data for their own purposes, regardless of the underlying data storage format. Using DAT requires no special knowledge about the content of the underlying data. DAT's advanced query language allows analysts to retrieve from the database exactly the portion of telemetry data that they need for their analysis. To provide flexibility, the query language has a large number of options and possible combinations of search and filter commands as specified in the query grammar. Users can query the system using a web-based query interface based on representational state transfer (REST) or through a web-based graphical user interface (GUI) to query the underlying PostgreSQL database, which stores metadata. The actual telemetry data is stored as binary data files. Query results are provided as data tables in several data file formats and as diagrams.

Since the DAT system handles spacecraft data that will be used for various forms of data analysis, the users of DAT must trust that DAT system is reliable. The users must also trust that the data it provides as a result of the users' queries is accurate and that the user queries are correctly handled. However, to achieve high confidence in the DAT system through the use of traditional testing methods is both tedious and error-prone. Traditional testing methods are primarily based on a large number of repetitive manual steps. Thus there is a risk that once the DAT system is released for production use, it is not systematically tested and therefore defects may remain undetected and slip into to the fielded system, only to be detected by the users at a time when it is likely to be costly to correct the mistakes.

The agile software development approach in use by DAT adds to the complexity of the testing process because in such an approach, requirements are allowed to change even late in the development process. Thus the testing approach for the DAT project must also support frequently changing requirements.

A. *Current testing practices*

The current testing practices do not support frequent change. The test cases of DAT are currently designed through a manual effort. That is, testers manually enter DAT queries and compare that the actual output matches the expected. This type of testing can only be conducted on a small scale and on very limited test data. One of the major problems, mentioned by developers of DAT, is that testing is not keeping pace with the agile development practices used in the DAT project. The result is that the developers must wait before getting feedback from the testers. Thus, in an agile project, testing becomes a bottleneck. It has also been observed in other projects that manual testing efforts often are uneven [7]; that is, some areas of a system are well-tested while other areas are not.

B. *Model-based testing in the context of agile methods*

Model-based testing (MBT) is a test case design technique for test automation. Instead of creating one test case at a time, the tester creates a model of the system under test, and lets the computer automatically derive test cases by exploring the model. These test cases can be automatically executed as part of the testing process. When a test case fails, a defect may have been detected. The model is a state machine that captures the externally visible states and the actions (or transitions) the user can trigger as well as the expected response from the system. The model is designed based on the tester's understanding of the system's requirements and system exploration, for example, by using and experimenting with the system.

In the case of DAT, the model is designed based on the specification of the documented grammar, which formally captures the query language DAT offers to the users. In this context, each test case represents one query and the comparison of the expected output with the actual output would provide the oracle. However, how to determine the oracle automatically is a problem that will be discussed below. When the requirements change as a results of the agile process, then the testing model is updated and new test cases automatically generated.

C. Model-based testing in the context of agile methods

Model-based test case generation is an appealing idea that is considered state-of-the-art, but is not yet used extensively by the software industry and therefore is not considered state-of-the-practice. MBT is promising, but the approach has not been adapted to agile projects and may have scalability issues if one attempts to model the entire implementation of the software under test. One issue is that the model may quickly become large and complex. In an agile project, it is important that the model can be updated in short time and therefore it must be well-organized and easy to understand and modify. Based on the successfully applied research conducted earlier [8][7][9], we instead set out to apply an MBT approach that focuses on the interface of the software under test. This limits the application of model-based testing to the input/output behavior, which can be described using a smaller model. Since a system can be broken down into an arbitrary number of subsystems or components, each with a defined interface, we have used this approach to gradually test the system in more detail. Even by adding details related to the interface-behavior of the system under test, the model still will only be a fraction of the size and complexity as compared to a model of the implementation.

The (testing) model that drives model-based testing describes the SUT's expected behavior as a sequence of stimuli and responses, as well as timing conditions, parameter values and parameter constraints. The model is precise yet agnostic of the many other details implemented in the source code and of the programming language. The benefit of this approach is that even though the system under test is large, the model describing it will remain manageable in size. This is a key in the context of agile projects. As a consequence, automatically generating test cases from this model and executing the test cases will also be manageable because of the reduction in size and complexity.

D. Automated testing of a query-based system

The DAT system can be divided into three logically separated layers: a data ingest layer, a data archive layer, and a data access layer. Testing as described in this paper focuses on the data access layer because it is the most critical part from a user's point of view. This comprises testing that the DAT API returns the correct data to the user, using happy (non-evil) testing methods. The reason we first addressed happy testing is that since the system is under development, the priority is to test that the functionality (new as well as old) of the system is working correctly. Once the system is stable, the testing objective will change to checking that the system is robust using evil testing. To achieve this, we have designed a MBT testing environment that includes modeling the query grammar, generation of REST queries as well as SQL queries (to establish the correct result: the oracle) and automatic comparisons of the results. If the comparison of the output from the two different queries return different results then this may indicate the detection of a defect in the DAT system, but would require further investigation since the deviation also could initially be the result of other inconsistencies between the expected and the actual results. For example, model defects are typically revealed during this analysis At this point, several versions of models of the DAT grammar have been developed in an iterative and incremental fashion to demonstrate the feasibility of the proposed approach to the NASA team.

Typically, MBT uses models with built-in test oracles (a.k.a. assertions). However, since the underlying data is unknown and very large, and since the DAT system has a relatively large query grammar, there are many different variants that need to be tested and thus the model needs to cover these grammar options and it is difficult to determine the oracle. We addressed this issue by using the data itself to create a separate testing database.

The telemetry data is stored in binary files called mnemonic archive data (MAD) files while information about the files and their location is stored in the Postgres database. To create the test database, we extracted all data and stored it in the separate testing database. The testing database is used to check that the result from the generated test queries is consistent with the specification. Thus, the test assertions will be consistent with the state of the database because the oracles will be based on the specifications and/or actual content of the database. Another benefit of this approach is that it facilitates replacing one database under test for another one because as soon as the new database under test has been installed, its content is transferred into the testing database.

We have also developed tools that extend the current prototypes for analyzing the runtime log files of DAT. These log files contain valuable information such as the DAT queries issued by the users and the corresponding SQL queries that are automatically generated by DAT. Using the log files, we can automatically check whether the conversion from the DAT queries to the SQL queries is consistent. Any deviation between the two would indicate an issue that might result in a detected defect. The database behind the DAT system contains data points associated with mnemonics. These data points have a specific time, raw value (raw), converted value (conv), and flags indicating the quality of the data; referred to collectively as value types (or vtypes).

E. Requests

Since the DAT web service adheres to the REST architectural style, its resources are identified by URIs (Uniform Resource Identifiers) and retrieved via the HTTPS protocol. Throughout this paper, these URIs are referred to as 'requests' and retrieving resources using them as 'requesting'. At the moment, the testing of the web service focuses on the data report segment of the web service. Data reports contain mnemonic names and their values extracted from the archive over some specified time range. Report requests start with the URI of the DAT host, followed by "/report", an optional format specifier (e.g., ".json"), a query string separator ("?"), and a query string. While the first part will be constant because we currently focus on the reporting function, the last part, the query string, will vary and is thus more interesting. A query string is composed of a series of value pairs. Within each pair, the field name and value are separated by an equals sign ("="). The series of pairs is separated by an ampersand ("&"). Fields are referred to as "selectors" by the DAT User's Guide. [6] Take, for example the complete but simple request "https://192.168.109.133/dat/report.csv?start-time=2010-12-10-23:00:38.7&stop-time=2014-12-24-10:12:10&mnemonics=MyMnemonic". The request starts with the location of the DAT web service, i.e. "https://192.168.109.133/dat", and then specifies that the result should be returned as a comma-separated values (csv) report before finally specifying the criteria for the data request as a query string, i.e. "start-time=2010-12-10-23:00:38.7&stop-time=2014-12-24-10:12:10&mnemonics=MyMnemonic". The fields of the query string and their respective values are "start-time", "stop-time", and "mnemonics"; and "2010-12-10-23:00:38.7", "2014-12-24-10:12:10", and "MyMnemonic". This means: return all data points for the instrument named "MyMnemonic" that have a time stamp that is equal to or greater than the start-time and equal to or less than the stop-time. Much more complex requests can be formulated since one can specify a list of mnemonics, flags, converted data, various forms of time intervals in several formats, data sampling, data aggregation, etc.

The DAT User's Guide [6] presents a context-free grammar that generates the formal language of legal request formats. The guide also specifies additional restrictions to be enforced. These restrictions are selector uniqueness, required selectors, and sensible value type selections.

III. Testing questions for DAT

Based on DAT's grammar and capabilities, we derived testing questions, displayed in table I, that must be investigated in order for the users to feel confident that DAT is working correctly.

TABLE I: TestingQuestions

Q1. Does the selector order matter? E.g. the query string "start-time=2010-12-10-23:00:38.7&stop-time=2014-12-24-10:12:10&mnemonics=MyMnemonic" should be equivalent to "mnemonics=MyMnemonic&stop-time=2014-12-24-10:12:10&start-time=2010-12-10-23:00:38.7".
Q2. Do different date formats matter? For example a query based on the format YYYY-DDD (e.g. 2014-001) should be equivalent to YYYY-MM-DD (e.g. 2014-01-01).
Q3. Do different date specifications matter? For example, if the underlying data set spans from 2010-12-10-23:00:00 to 2010-12-15-21:00:00 (i.e. there is a data point with the time stamp 2010-12-10-23:00:00 and one with the time stamp 2010-12-15-21:00:00), then a query with a start time before and a stop time after these data points should be equivalent to a query with a start and stop time exactly on these time stamps.
Q4. Does the number of date parts matter? E.g., "duration=15d5h3s" should result in different results than "duration=15d".
Q5. Do value types specified on the mnemonic level properly take precedence over ones specified on the selector level? E.g., does the DAT web service know to serve the flag value type instead of the raw value type when presented with the request "report?mnemonics=MyMnemonic(flag)&vtype=raw&start-time=2010-12-10-23:00:38.7&stop-time=2014-12-24-10:12:10".
Q6. Does DAT return the correct value types for each data point? Vtype can be specified for each mnemonic and/or for the entire query. In case several Vtype specifications are provided, a rule set determines which ones take precedence. The rule set specifies that vtypes specified for each mnemonic take precedence and that the settings "vtype=all,raw,flags,conv" is equivalent to "vtype=all" and that multiple mentions of the same parameter, e.g. "vtype=raw,raw,raw,flags" is eqvivalent to "vtype=raw,flags" etc.
Q7. Are the time stamps of the data points DAT returns correct?
Q8. Are the values DAT receives interpreted correctly?
Q9. Does DAT return the correct number of data points (number of rows as well as number columns)?
Q10. Does DAT return the correct data points?

Please note that the different numbers of rows and columns, and different individual data items are indicators of issues. Thus, the answers to the testing questions can be determined by analyzing test cases that fail because the test results differ.

IV. Testing Framework

The DAT system and the web service it provides was delivered to the tester on a virtual machine capable of running it. Hereinafter, this particular virtual machine will be referred to as 'NASAVM'. This is the standard method of delivery and, since several versions were anticipated, being able to replace the NASAVM with the latest version had to be straightforward. Setting the testing framework up on the NASAVM was therefore unacceptable; replacing the NASAVM would require setting the entire testing framework up anew. To minimize the setup time required to get a new version up and running, a second virtual machine was created and dubbed

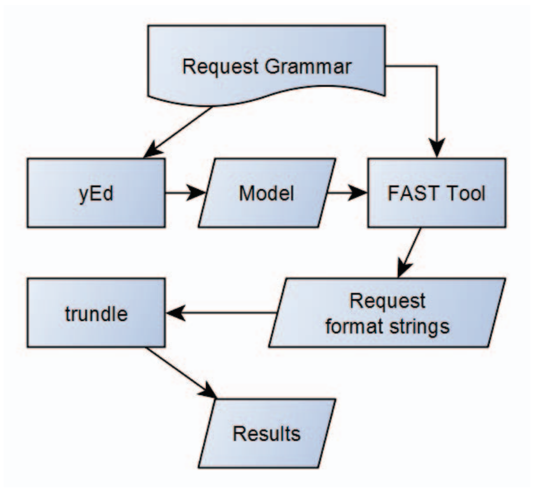

Fig. 1: The testing framework. yEd is used to create a model from the request grammar. The FAST Tool uses said model along with the request grammar to generate request format strings. `trundle` then uses those format strings to test the DAT web service.

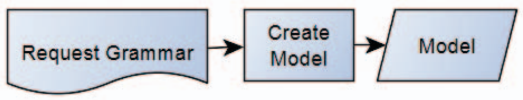

Fig. 2: The yEd related portion of the framework. This graph is an elaboration on yEd and its inputs/outputs in figure 1.

'FraunhoferVM'. The purpose of the FraunhoferVM was to provide a testing framework that was decoupled from the system under test. [1]

The testing framework consists of three components: yEd, the FAST Tool, and `trundle`.

A. yEd

yEd [2] is a graph editor that was used to model the request grammar (II-E). Hereinafter, this is the model which is being referred to when 'the model' is referenced. An effort was made to keep the model faithful to the grammar—more often than not, a node in the model corresponds with a symbol in the grammar. A node not corresponding to a symbol is in most cases one that simplifies the model considerably. Take, for example, the 'start' and 'stop' nodes in figure 3; they indicate where in the Selectors group node traversal should begin and end. In this case, using those nodes as a rendezvous simplifies the model considerably. Guards on the edges connecting the nodes enforce the additional restrictions mentioned in II-E. For example, an edge leading to a node corresponding to a selector 's_1' in the grammar will be associated with a Boolean value 'b_1'. Before this edge is traversed, b_1 has been initialized as false. Traversing the edge leading to s_1 requires b_1 to be false. After traversing the edge, b_1 is changed to true. Thus selector uniqueness is achieved for s_1. Figure 3 provides an overview of how this works.

[1]FraunhoferVM used the CentOS operating system, but all modules of the testing framework should work on a computer running UNIX/Windows. CentOS was chosen because NASAVM ran it.

[2] http://www.yworks.com/en/products/yfiles/yed/

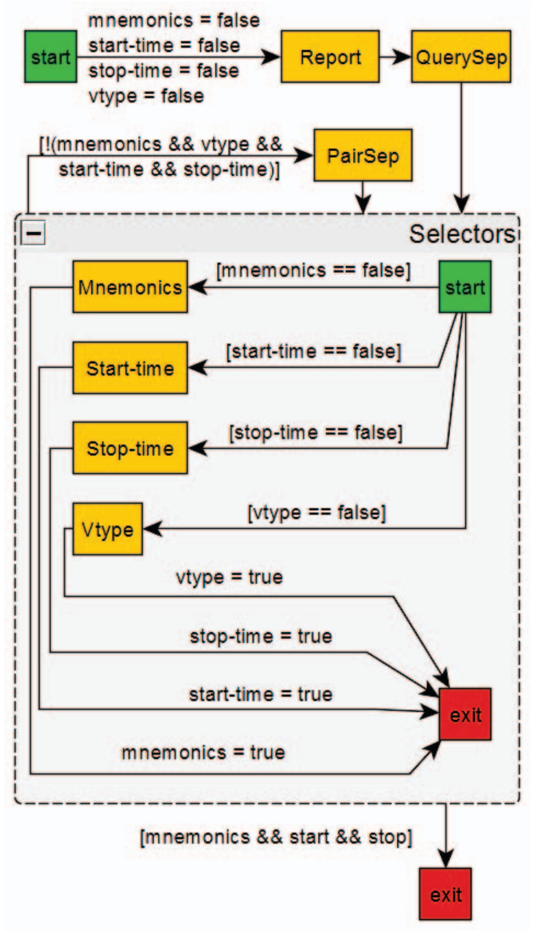

Fig. 3: A simplified version of the model used in the testing framework. In this simplification, Mnemonics is just a single node, while in the actual model it looks like figure 4. The Start-time, Stop-time, and Vtype nodes are similar simplifications. The code in brackets are guards that must evaluate to true for the edge to be traversable. These guards along with the other code on the edges take care of the additional restrictions. The most notable restrictions enforced in this simplification are the required selectors and selector uniqueness ones.

B. The FAST Tool

The FAST Tool (Fraunhofer Approach to Software Testing) is a tool developed at Fraunhofer CESE that enables a straightforward approach to model based testing. The tool is based on graphwalker, which is "a Model-Based testing tool built in Java. It reads models in the shape of finite-state diagrams, or directed graphs, and generate[s] tests from the models, either offline or online." [3] The FAST Tool allows the user to manage, maintain, and visualize (e.g., test coverage) their MBT projects through a user interface. The FAST Tool is used to create abstract test suites; it uses graphwalker to

[3]http://www.graphwalker.org/

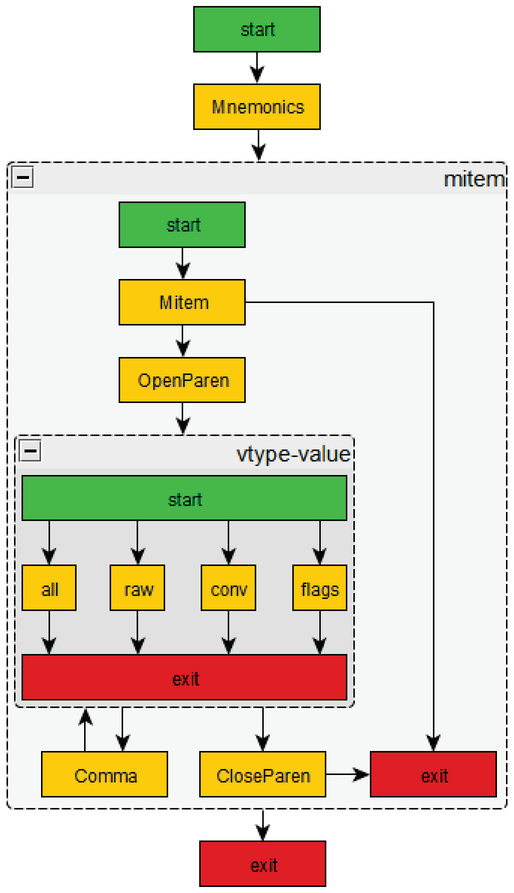

Fig. 4: The mnemonics portion of the model.

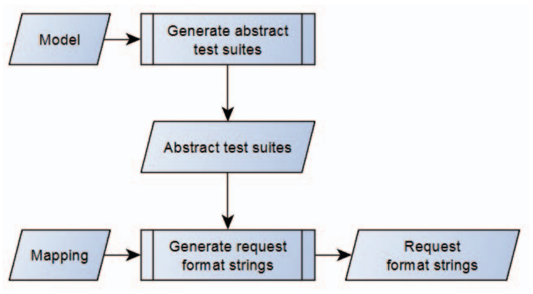
Model
Generate abstract test suites
Abstract test suites
Mapping
Generate request format strings
Request format strings

Fig. 5: The FAST Tool related portion of the framework. This graph is an elaboration on the FAST Tool and its inputs/outputs in figure 1.

traverse the model made with yEd using various traversing algorithms. An abstract test case is a traversal through the model, starting at the start node and ending when the stopping criteria is met. The user can specify how many such abstract test cases to generate, as well as a stopping criteria such as traversal length and state (node) or transition (edge) coverage. The traversal will stop when the stopping criteria is met or if the exit node has been reached, depending on which one occurs first. The latter is enforced in this testing since it is not to be evil. Each traversal results in an abstract test case which is a list of visited nodes in the model. These test cases are abstract because information is still missing for them to be executable (concrete). In order to add that information, a mapping is used. A mapping is a function that takes a node or edge label as input and replaces it for something more specific or concrete. For testing of DAT, the FAST Tool has been configured to translate each traversal into a request format string. This is done by translating each abstract test case into `python` scripts that produce the request format strings. The mapping process takes as input the set of nodes and edges corresponding to terminal symbols in the grammar while the outputs are the corresponding terminal symbols.

Some terminal symbols, such as the value associated with the mnemonics selector, require more knowledge of the data DAT can serve and is therefore determined at a later stage in the process. Those values were mapped to a placeholder to later be replaced by `trundle`.

For an example of a traversal of the model in figure 3, take "Report → QuerySep → Selectors → Mnemonics → PairSep → Selectors → Start-time → Selectors → Stop-time". Using the example mapping in table II, its resulting request format string would be "report?start-time={start}&mnemonics={mnemonics}&stop-time={stop}". Here "{start}", "{mnemonics}", and "{stop}" are the placeholders for values to be inserted later.

TABLE II: MappingExample

Node	String
Report	report
QuerySep	?
Selectors	
Mnemonics	mnemonics={mnemonics}
Start-time	start-time={start}
Stop-time	stop-time={stop}
Vtype	vtype=all
PairSep	&

C. `trundle`

`trundle` [4] is the workhorse of the testing framework and its task is twofold: create test cases based on request format strings generated by the FAST Tool; and run tests cases. It was written in `python` and amounts to 634 lines of code excluding comments and empty lines. The common test code (setup and teardown as well as helper functions) was structured in such a way that each test case only contains the unique code. Self-containedness and determinism was also considered. For the test cases to be deterministic, they are populated with concrete values (e.g., start and stop times) instead of generating them at runtime. For them to be self-contained, they would have to use no information outside of their code (which would mean that passwords, for example, would be hardcoded). The solution

[4]The name '`trundle`' was inspired by how slowly the first working version of the script ran.

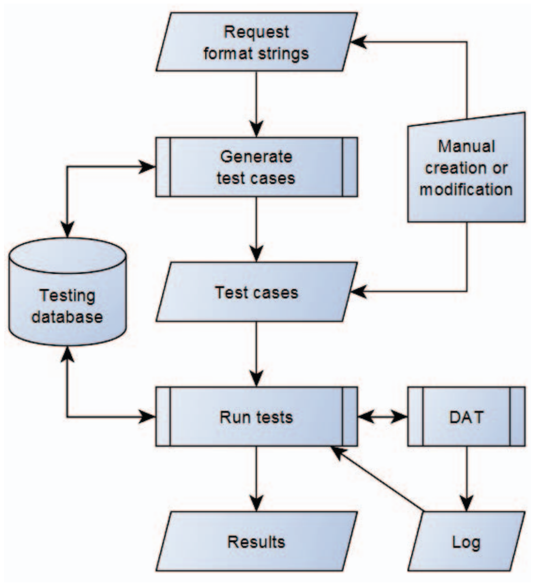

Fig. 6: The `trundle` related portion of the framework from figure 1.

was to make the abstract test cases self-contained and thus they can be sent to the NASA team independently of the underlying data. The concrete test cases are dependent on the content in the underlying database and can be used directly as long as the content of the underlying database is the same.

When the DAT system receives a request, it logs the process of responding to that request to a development log on the NASAVM. `trundle` uses the Secure Shell (SSH) protocol to access the NASAVM and read this logfile. It uses the logfile information to determine if DAT is correctly parsing and casting the values given to its web service.

To verify a response from the DAT web service to be correct, `trundle` had to be able to infer the correct response. For this purpose, the DAT system was first queried for all the data points for a subset of all possible mnemonics and aggregated into a PostgreSQL database on the FraunhoferVM. `trundle` queries this local database—hereinafter referred to as the testing database—and use the result table along with the DAT request, to infer the expected result. Doing this meant that assumptions had to be made on the correctness of the results to the queries used to make the testing database. However, since all test case failures revealed true issues, this assumption seems to hold. At this point the testing process is independent of the model and the previously mentioned tools—the output of the FAST Tool can be used by `trundle` to run and re-run test suites without the involvement of the model. Since the test cases produced by the FAST Tool contain DAT request format strings, the value placeholders in those strings—namely, the value placeholders associated with time-related selectors and the value placeholder associated with the mnemonics selector—need to be replaced with actual values in the first run by `trundle`. For example, the mnemonics placeholder is replaced with concrete mnemonic values. Each request format string is used to create one test containing multiple concrete requests (test cases) which differ on the values for the time-related selectors. and for the request to be legal, stop-time has to be greater than or equal to start-time. To get a thorough coverage, a set of times T' is constructed and $T \subset \{(a, b) \mid a \in T', b \in T', a < b\}$ is used as the set of pairs of start-time and stop-time; that is to say, a subset T of the pairs formed by using elements from the set T' with the additional requirement that the former element in the pair must be less than the latter, is used. [5] T' is constructed by starting with an empty set and adding four times: the earliest legal time (i.e., 1970-01-01-00:00:00.0), the time of the earliest data point for the mnemonic in question, the time of the latest data point for the mnemonic in question, and the latest legal time (i.e., 2035-01-01-00:00:00.0). Then two random times are added in-between every two chronologically adjacent pairs of these four. Some additional pairs were pruned out due to them being comparable to other pairs. For example, when using the two in-between times a and b, where $a < b$, as start times, b produces pairs with the same stop-times as a (excluding a's first pair which will be the pair a, b). Since a and b are both random values in the same time interval, b can be dismissed as a start time.

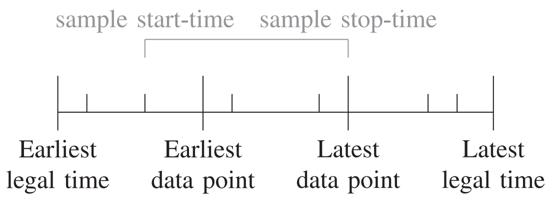

Fig. 7: The horizontal rule represents all possible legal DAT time values, going from early (left) to late (right). The vertical lines on the rule are all the times from the set T'. The longer vertical lines at the ends of the rule represent the earliest and latest legal times. The other two longer vertical lines on the rule represent the earliest and latest data points for a specific mnemonic. The smaller vertical lines on the rule represent the randomly generated in-between time values. The interval on top of the ruler is an example of an interval in T; e.g., a start-time, stop-time pair.

Once these time stamps have been established they are used to replace their respective placeholders in the request format string. This results in 24 concrete requests since the cardinality of T in the current implementation of `trundle` is 24. These requests are saved to a new folder, referred to as the 'test' generated for that request format string, in the test suite and the file containing the original request format string is deleted. This process is never repeated for a test suite because it is the process of taking request format strings and using randomness to replace its placeholders. Once this is done for all the tests, `trundle` is left with a test suite containing concrete DAT web service requests.

To reiterate in a more concise manner: When `trundle` runs a test suite, generated by the FAST Tool, for the first

[5]Since it's perfectly legal for start-time to be equal to stop-time, the last predicate will be $a \leq b$ in future versions of `trundle`.

time (and not in subsequent runs!), it does the following for each request format string:

1) Randomly select a mnemonic in the testing database.
2) Heuristically generate a distinct set T of 24 intervals.
3) Use the selected mnemonic and the 24 intervals in T along with the format string to build 24 concrete DAT web service requests.
4) Save the resulting requests (test cases) in a distinct folder, referred to as the 'test' generated for that request format string, in the test suite.
5) Delete the original format string.

At this point the test suites are concrete and deterministic. Given that the DAT web service is not changed in such a way that sacrifices backwards compatibility of requests and that the testing database is up to date, they can be run repeatedly with the same results. Say, however, that a selector's name is changed in the future, then the test suites need to be scrapped or manually edited to work. In both cases, `trundle` needs to be modified accordingly so it can identify the changed selector. Since the testing process is to be as automated as possible, manually modifying all test suites is out of the question. Thus, the proper response to the selector's name change is to scrap the test suites. However, the abstract test suites remain and, by modifying the mapping to take the name change into account, new test suites can be generated to replace the scrapped ones. Those test suites start out as request format strings again and need to be converted into tests again by `trundle`. However, there is no guarantee—and in fact highly unlikely—that the values generated/selected by `trundle` will be the same this time as when it generated/selected them before. Thus the test suites will no longer be identical to the old ones.

Now, and in subsequent runs for this suite, `trundle` runs the tests. The CaseReport class is a container class for information on a test case and has a method to verify DAT results. To enable the verification of a CaseReport, it has to be populated with a) the response from the DAT web service; b) the result table from the testing database; and c) the relevant lines from the development log.

For each test in the test suite `trundle` creates a list of CaseReport objects and aggregates the information for them in the following manner:

1) Query the DAT web service with a request in that test and insert into an exclusive CaseReport.
2) Query the testing database for the same mnemonic and interval as the DAT request and insert the resulting result table into the same CaseReport.
3) Repeat the steps above until no more test cases remain.
4) Get the lines from the development log on the NASAVM corresponding to the requests made and add those to their respective CaseReports.

The last step is done after all of the requests have been made for the sake of efficiency. When the list of CaseReports is ready, `trundle` verifies each test case one after the other (make assertions on the results from DAT). The verification method verifies the following:

- The values in the development log: start-time, stop-time, flags, mnemonics, etc. These are compared with the values in the request to see if DAT is processing those properly.
- That the number of returned data points is correct.
- That the time of each data point is correct.
- That the returned value types are correct.

If at any point in this process an error is raised, such as OSError (indicating a system-related error), or an assertion fails, `trundle` stops the test and continues with the next one. Both errors and assertion failures are reported to the tester and, if the verbose option (-v or –verbose) is set, details on the failed assertion or error is printed (e.g., `Failed test "3" case "9": Number of data points does not match expected value`). If no error or assertion is raised, the test has passed.

V. Issues discovered

A. *Issues discovered by automated testing*

The testing framework was used to generate seven test suites; seven for no better reason than it is a small and manageable number. The FAST Tool was used to create the seven abstract test suites, each containing 100 test cases where each test case is one parameterized query or a request format string. The tester decided to create 100 abstract test cases in each suite because it seemed to be a reasonable starting point. Then each abstract test suite's 100 abstract test cases were instantiated as 100 request format strings in physical test suites; i.e. as (request format) strings in files in directories/test suites. When `trundle` ran these test suites for the first time, it did the following for each format string:

1) Randomly selected a mnemonic from the mnemonics in the testing database.
2) Generated a distinct set T of 24 intervals.
3) Used the selected mnemonic and the 24 intervals in T along with the format string to build 24 concrete DAT web service requests.
4) Saved the resulting requests.
5) Deleted the original format string.

This made the test suites concrete and deterministic; it also made for a total of 16,800 DAT requests for `trundle` to run and verify. [6] Surprisingly, all of the tests in all of the test suites failed the same assertion (unexpected number of data points returned by DAT), and most of them on the same case; that is, the same type of date interval. Manual analysis of the failed tests revealed two issues:

1) Segregation of Ultimate Data Point: About 90 percent of the test cases failed when a single data point comprised the expected response whereas the response from DAT contained no data points at all. Further inspection revealed this case to be the one where the starting time of the request's time interval

[6]Assuming each test case had to be a self-contained `python` script and contain all `trundle` related code (about 25KB), these seven test suites would have amounted to 420 Megabytes!

was that of the latest data point for its mnemonic; the very same data point which was expected but missing. [7]

During the analysis of the failure, which included augmenting the test suites by crafting additional tests, provided analogous, albeit surprising, results. The entirety of the testing database consisted of mnemonics where the ultimate data point fell on a time specified with 500 milliseconds. Some of the manually constructed tests specified start times that fell around and on these; e.g. 0, 1, 345, 499, 500, 501, etc. milliseconds. These revealed that the ultimate data point was not introduced to the DAT response until the start time of the request reached zero milliseconds. Other tests supported the assumption that this behavior was unique to the ultimate data point by repeating the process with other data points.

2) Disrupting Value Type: About 10 percent of the test cases failed faster than others on the same assertion (unexpected number of returned data points). At a first glance this behavior appeared to be triggered by the stop time of the requests' time interval being equal to their missing data point; the first data point of a mnemonic. This would have made the issue a complement to the previously disclosed issue, but could not explain why all the tests did not fail this case. Further inspection revealed that the failed case was but first of many, in fact, no response from DAT contained any data points at all. It turned out that if the only value type requested was flags, DAT would respond with an empty set of data points and as if no value type was specified.

B. Issues discovered in development of the testing framework

During the development of the testing framework, some manual—or partly manual—exploratory testing was done in the context of affirming that the framework was working as intended. This testing revealed several issues which can be viewed as having been discovered by automated testing because each was either a) discovered so late in the development of the automated testing framework that its discovery can be attributed to it; or b) provably detectable by the automated testing framework. In fact, during development, these issues had to be dismissed so the developers could see anything other than failed tests.

1) Dysfunctional Duration Selector: The request grammar defines how the value associated with the duration selector is supposed to be specified. For example, "14d15h16m" and "14d" are two perfectly legal values that could return different results. However, due to a detected issue, these values actually produce the same result because internally DAT discards everything after the first portion of the value string; e.g. since "14d" is the first portion, the rest "15h16m" is discarded. Thus, both "14d15h16m" and "14d" are incorrectly evaluated to fourteen days. This is unintended behavior that could result in incorrect results and was as such reported as an issue.

[7]Early on in this project, T only consisted of the earliest and latest legal times and random times in-between those. This issue was unlikely to surface using that T so it is clear that the design of T is important.

2) Unhandled Leap Seconds: "A leap second is a one-second adjustment that is occasionally applied to UTC [. . .] in order to keep its time of day close to the mean solar time." [10]

Since DAT is a system that handles spacecraft telemetry, for which correctly handling time is of utmost importance, handling leap seconds is critical. However, our testing revealed that the tested version of DAT lacks leap second support. For example, if DAT is queried with a request that specifies a perfectly legal UTC start-time value of "2012-06-30-23:59:60", the leap second is dropped by DAT; the value is treated as "2012-06-30-23:59:59". This could result in incorrect data being returned and was thus reported as an issue.

3) Unusable Delimiters: The request grammar specifies three delimiters for the UTC time strings ('T', '-', and ' ') but only one of them—the dash—is implemented and usable. All other formats return incorrect results.

C. Results of evil testing

Albeit unintentional, some evil testing was performed and issues discovered by this testing are discussed here.

1) Silent Time Parsing Failure: Omitting, for example, the seconds or specifying the 30th hour a UTC time string results in a silent error; DAT falls back on the default time of "00:00:00.000". The web service thus, for example, treats both "2012-06-30-23:59" and "2012-06-30-155:50:23" as "2012-06-30-00:00:00.000". The error being silent is of most concern here, no mention is made of a faulty or truncated time string, and so it was reported as an issue.

2) Irrational Time Values: Specifying start-time and stop-time such that start-time is less than stop-time does not result in an error. This is something that may be prudent to notify a user of and was thus reported as an issue.

VI. Answers to DAT testing questions

Based on the described testing of DAT, we arrive at the answers in table III. The table reveals that 6 of the 10 equivalence based testing questions resulted in detected defects. For example, even though it should not matter how dates were specified, the query results did, at times, differ for this reason. See table I for sample queries that should be equivalent.

VII. Related Work

There are a few technical papers that are related to testing a system in which it is difficult to formulate the test oracle.We discuss those papers from the perspective of our work.

A. Metamorphic testing

In metamorphic testing [1] the test oracle problem is alleviated by formulating one or more properties about the SUT using so-called metamorphic relations between inputs and outputs (e.g. $\sin(-x) = -\sin(x)$). In our approach, we apply the idea of metamorphic testing based on properties of the data stored in the database of the SUT as well as the properties of the grammar and the SQL test queries (e.g. equivalent queries).

TABLE III: TestingAnswers

Q1. Does the selector order matter? No issues detected.
Q2. Do different date formats matter? No issues detected.
Q3. Do different date specifications matter? Issues detected.
Q4. Does the number of date parts matter? Issues detected.
Q5. Do value types specified on the mnemonic level properly take precedence over ones specified on the selector level? No issues detected.
Q6. Does DAT return the correct value types (vtypes) for each data point? Issues detected.
Q7. Are the time stamps of the data points DAT returns correct? No issues detected.
Q8. Are the values DAT receives interpreted correctly? Issues detected.
Q9. Does DAT return the correct number of data points (number of rows as well as number columns)? Issues detected.
Q10. Does DAT return the correct data points? Issues detected.

B. *Differential testing*

Differential testing compares the behavior of a tested system to another implementation with similar functionality [11]. Every divergence, in theory, exposes an error in either the tested system or the reference system. Their approach randomly invokes the system under test and compares the actual output with a reference system implementation. In contrast, we built the model of the grammar to generate queries automatically; the test verdict is based on comparing the records returned by the SUT to the records returned by our queries that are sent to the testing database. Our reference implementation is the development of separate queries directly sent to the database.

C. *Model-based testing*

There are many MBT case studies in the literature. For example, Harry Robinson has created a website on MBT [12] with a collection of papers, presentations, etc. We have applied MBT on different types of systems [8][7][9]. In contrast to the existing MBT literature, this work uses MBT to generate two types of queries without any test oracle embedded in the model. One query is based on the grammar of the SUT and the other corresponding query is in SQL. The test verdict is based on comparison of the records returned by the two syntactically independent queries.

Conclusions

We have described how we designed, implemented, and applied a framework for automated testing of NASA's DAT system. One of the main testing challenges is full scale acceptance testing of DAT because of the problem to determine whether or not the results DAT provides are correct. We addressed this challenge by using an approach where we use the system and the underlying data to test itself. Queries are automatically generated from a model and equivalent variants of the queries are generated and executed in different ways. The results, that should be equivalent for the equivalent queries, are compared. Deviations indicate issues. The approach has detected a number of confirmed issues and therefore is deemed a promising approach. The issues have since been addressed. The architecture of the testing framework has been designed with flexibility in mind in order to facilitate testing of new versions as they arrive. Our experience so far is that the built-in flexibility has been useful and supportive of this type of testing in a project that uses agile methods. We believe the same approach can be used on other systems that have the same problem with defining an oracle. Since the DAT project is still ongoing, we will continue developing and applying the approach to conduct acceptance testing.

Acknowledgments

This work is supported by NASA's Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP). We especially thank LaMont Ruley, Kenneth D. Rehm, Ricky Forquer, Wesley W. Deadrick, Markland Benson and Martha Wetherholt for their continuous support. Haukur Steinn Logason developed the testing infrastructure and Christoph Schulze provided support.

References

[1] 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

[2] T. Clune and R. Rood, "Software testing and verification in climate model development," *IEEE Softw.*, vol. 28, no. 6, pp. 49–55, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1109/MS.2011.117

[3] harryr@google.com, "Exploratory automation." [Online]. Available: http://www.harryrobinson.net/ExploratoryTestAutomation-CAST.pdf

[4] M. Lindvall, D. Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May, and T. Kähkönen, "Agile software development in large organizations," *IEEE Computer*, vol. 37, no. 12, pp. 26–34, 2004. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MC.2004.231

[5] D. Cohen, M. Lindvall, and P. Costa, "An introduction to agile methods," *Advances in Computers*, vol. 62, pp. 1–66, 2004. [Online]. Available: http://dx.doi.org/10.1016/S0065-2458(03)62001-2

[6] *Data Access Toolkit (DAT) User's Guide*, 2014, version 0.8.

[7] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: An empirical study conducted in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 135–144. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591180

[8] C. Schulze, D. Ganesan, M. Lindvall, D. McComas, and A. Cudmore, "Model-based testing of nasa's osal api - an experience report." in *ISSRE*, 2013, pp. 300–309.

[9] 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.

[10] Wikipedia, "Leap second — wikipedia, the free encyclopedia," 2014, [Online; accessed 23-October-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Leap_second&oldid=630477389

[11] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 621–631.

[12] H. Robinson, "Model-based testing." [Online]. Available: http://www.harryrobinson.net/