Detecting Inconsistencies in Wrappers: A Case Study

Henning Femmer
Technische Universität München*
Munich, Germany
femmer@in.tum.de

Dharmalingam Ganesan, Mikael Lindvall
Fraunhofer CESE
College Park, Maryland, USA
{dganesan, mlindvall}@fc-md.umd.edu

David McComas
NASA Goddard Space Flight Center
Greenbelt, Maryland, USA
david.c.mccomas@nasa.gov

***Abstract*—Exchangeability between software components such as operating systems, middleware, databases, and hardware components is a common requirement in many software systems. One way to enable exchangeability is to promote indirect use through a common interface and an implementation for each component that wraps the original component. As developers use the interface instead of the underlying component, they assume that the software system will behave in a specific way independently of the actual component in use. However, differences in the implementations of the wrappers may lead to different behavior when one component is changed for another, which might lead to failures in the field.**

This work reports on a simple, yet effective approach to detect these differences. The approach is based on tool-supported reviews leveraging lightweight static analysis and machine learning. The approach is evaluated in a case study that analyzes NASA's Operating System Abstraction Layer (OSAL), which is used in various space missions. We detected 84 corner-case issues of which 57 turned out to be bugs that could have resulted in runtime failures.

***Index Terms*—Abstraction, Equivalence, Inconsistencies, Interfaces, Machine Learning, Wrappers**

I. Introduction

Being able to exchange one software component for another is a requirement for many software systems. One way to make components exchangeable even though they have different interfaces is to define a common *interface* and *implementations* of this interface (wrappers) that hide the details of each underlying component (see Fig. 1). Examples of such *software abstraction layers* (SALs) are hardware abstraction layers, database abstraction layers, algorithmic abstraction layers, middleware abstraction layers, and *OS abstraction layers* (OSALs) [1], [2].

In this paper we study the use of OSALs, particularly the OSAL that was developed by the NASA *Core Flight Software System* (CFS) team at Goddard Space Flight Center. Yet, the same reasoning applies to all types of SALs, hence we believe that many practitioners would be interested in this study.

In systems using an OSAL, developers program to the interface instead of directly to a particular OS. Therefore, they trust that it does not matter which OS is in use during execution. Consequently, they run into problems if this is not the case. For example, different OSes typically produce different error codes. If the error codes from different OSes

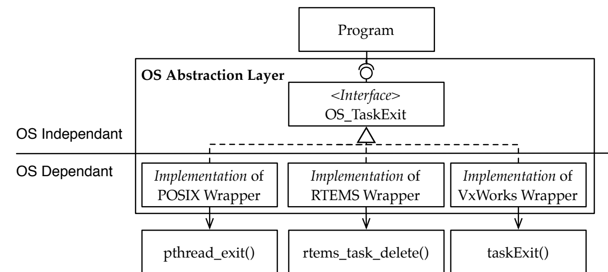

Fig. 1. Example OS Abstraction Layer for the function `OS_TaskExit`

are not "wrapped" in the same way, then the same input can result in different error codes, and thus different behavior of the calling program. Accordingly, the OSAL must correctly translate the error codes from each individual OS into a common set of error codes. It is the responsibility of each (wrapper) implementation to provide this translation so that the differences between the underlying OSes remain unexposed to its users. Since translation defects can alter the perceived behavior of the underlying OS (i.e. the OS behaves correctly but wrapper-defects distort the visible behavior), it must be verified that all implementations are indeed equivalent. The same principle applies to all aspects of the interface, i.e. on equal input the implementations should have equal output.

One verification method is testing. However, since a system such as CFS has a huge number of configuration options, it is extremely difficult to test all possible combinations. Thus, complementary defect detection strategies are of interest to the CFS team. This led us to study whether an approach based on basic heuristics and easy-to-collect source code metrics would serve as a complementary verification method. The method is based on the fact that all equivalent wrappers must handle the same type of situations (e.g. if one wrapper handles three different error scenarios, we expect all equivalent wrappers to handle three error scenarios (no more, no less)). They must also make similar decisions (e.g. if one wrapper makes four decisions, we expect all equivalent wrappers to make four decisions). Thus, the key idea is to collect data about return codes, string constants, number of conditional statements, etc. for each implementation. These sets of data are compared to each other to detect differences so that the analyst can review the differences for possible issues. In addition, since there are more than 100 functions that are implemented three times,

*Henning Femmer was at Fraunhofer CESE while conducting this work.

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Software Engineering in Practice

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

once for each OS, we also wanted to study if machine learning could be used to automatically detect issues. The approach is practical and attempts to quickly detect selected issues related to equivalence. The results show that the approach has the potential to detect non-trivial bugs.

A. Contributions

This paper makes the following contributions:

1) We describe a new approach to determine whether two implementations are equivalent. The approach uses a) lightweight static analysis for extracting source code metrics, constants, etc. to compare two implementations and b) machine learning to automatically classify the differences and predict whether or not the implementations are equivalent.
2) We present a catalog of bugs that are present in a SAL. Researchers may find this catalog useful for developing and evaluating techniques to detect such defects.
3) We analyze our approach w.r.t. two different research questions: RQ1: Which type of defects can we find with our approach? RQ2: To what extent can machine learning classify functional equivalence in OSAL?
4) We provide a case study on a real world OSAL that resulted in 84 issues and a prediction accuracy of 88%.

B. Structure of Paper

The rest of the paper is structured as follows: Section II describes the type of equivalence we focus on. Section III describes the approach that enables an analyst to identify various violations of equivalence. Section IV describes the case study that evaluates the applicability of the approach. Section V discusses the OSAL development context in light of the detected bugs. The last two sections summarize related work and conclude the paper.

II. Equivalence of Software Abstraction Layers

In order to understand what kind of differences we are analyzing, it is useful to first define in what way two implementations of a SAL may or may not be equivalent.

A. Defining Functional Equivalence for SALs

In software systems that support several different implementations of the same interface, these implementations must be exchangeable in a transparent fashion. This means that neither the developer nor the end user should notice a difference in behavior if the system is switching one implementation for another. If the system faithfully adheres to this architecture one can exchange one component for another without rewriting any application code. A SAL, which in this context consists of an interface and a set of implementations, can be used to achieve this. The interface specifies the functions of the SAL, and each implementation implements this interface in a different way, yet all implementations should be functionally equivalent. Functional equivalence means that if two implementations are given equivalent input, including parameters as well as global system state, both components should generate equivalent output, which is equal return codes, equal modifications of parameters and equal modifications of the global state [3]. However, determining whether two functions are functionally equivalent is a special case of Rice's theorem [4], which states that every non-trivial property (a property which holds at least for one, but not for all functions) is undecidable. Thus, no algorithm can decide whether or not two arbitrary implementations behave equally on all inputs and outputs. Instead, any approach can only achieve an approximation to functional equivalence within a limited scope.

B. Example: `OS_TaskExit`

The example (and the study below) is based on NASA's OSAL, which is used in space missions such as the Solar Dynamics Observatory (SDO) and the Lunar Renaissance Orbit (LRO) [5]. In these applications reliability is critical and hence NASA's OSAL has very high quality requirements [5]. It also provides a common interface for various system calls, allowing developers to use the same code base for different OSes. It wraps POSIX, VxWorks6 and RTEMS and can be easily extended to include other OSes as well. The OS to use is selected at build time.

As an example, two different implementations of the OSAL function `OS_TaskExit` that are intended to be functionally equivalent are shown in Fig. 2. The function `OS_TaskExit` is typically used by application developers to terminate the current thread. The interface is implemented for each supported OS so that users of OSAL can exit tasks without worrying about which platform the system is running on. However, when comparing the two implementations, we see syntactic code differences and especially three differences that may threaten functional equivalence and must be further investigated.

First, the RTEMS implementation declares an additional local variable called `status`. However, that variable is only written, but never read. Hence, neither control flow, nor external data is changed because of this difference. Second, POSIX and RTEMS have different ways to protect and free semaphores. In POSIX this is achieved by calls to `pthread_mutex_lock` and `pthread_mutex_unlock`, while in RTEMS this is achieved by calls to `rtems_semaphore_obtain` and `rtems_semaphore_release`. In addition, different parameters are passed to the respective POSIX and RTEMS functions. As these system calls are specified in the underlying OS, we do not know their internal behavior. However, assuming that the underlying RTEMS and POSIX functions are functionally equivalent, these differences have no impact on the external behavior of `OS_TaskExit`. Third, the field `id` in table `OS_task_table` is assigned at different locations in the code. Such differences may lead to semantic differences; however, it does not in this case since the variable is not used between the two locations.

Thus, although the implementations vary syntactically, we conclude that the two different implementations are equivalent from the point of view of the user of `OS_TaskExit`. We will describe examples where equivalence of SALs is violated in the evaluation in Section IV.

```c
..\..\..\Data\osal-3.2\src\os\posix\osapi.c
void OS_TaskExit()
{
    uint32 task_id;


    task_id = OS_TaskGetId();

    pthread_mutex_lock(&OS_task_table_mut);

    OS_task_table[task_id].free = TRUE;

    strcpy(OS_task_table[task_id].name, "");
    OS_task_table[task_id].creator          = UNINITIALIZED;
    OS_task_table[task_id].stack_size       = UNINITIALIZED;
    OS_task_table[task_id].priority         = UNINITIALIZED;
    OS_task_table[task_id].id          = UNINITIALIZED;
    OS_task_table[task_id].delete_hook_pointer  = NULL;


    pthread_mutex_unlock(&OS_task_table_mut);

    pthread_exit(NULL);

}
```

```c
..\..\..\Data\osal-3.2\src\os\rtems\osapi.c
void OS_TaskExit()
{
    uint32          task_id;
    rtems_status_code status;

    task_id = OS_TaskGetId();

    status = rtems_semaphore_obtain(OS_task_table_sem, RTEMS_WAIT, RTEMS_NO_TIMEOUT);

    OS_task_table[task_id].free  = TRUE;
    OS_task_table[task_id].id = UNINITIALIZED;
    strcpy(OS_task_table[task_id].name, "");
    OS_task_table[task_id].creator          = UNINITIALIZED;
    OS_task_table[task_id].stack_size       = UNINITIALIZED;
    OS_task_table[task_id].priority         = UNINITIALIZED;

    OS_task_table[task_id].delete_hook_pointer  = NULL;
    status             = rtems_semaphore_release(OS_task_table_sem);

    rtems_task_delete(RTEMS_SELF);

}
```

Fig. 2. Two equivalent implementations of `OS_TaskExit` (POSIX left, RTEMS right)

III. The Approach

Our approach can be seen from an analyst's perspective in the following five steps (see Fig. 3):

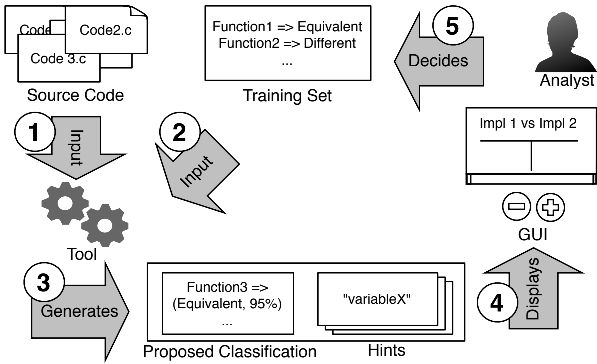

Fig. 3. The process of application

(1) The analyst imports the source code files that constitute the SAL to the tool. The functions to be compared for equivalence have identical signature, i.e. name, parameters and return code. Two functions of the same signature form a *function pair*. The tool detects differences in the function pair and highlights them to the user. In the example in Fig. 2, the differences are related to variables (e.g. the variable `status`), and function calls, including parameters passed to the function.
(2) The analyst notes detected defects and classifies a number of these function pairs, which are added to the initial training set. The training set consists of function pairs and a flag indicating whether the two functions are *equivalent* or *different* and is empty at the beginning, as all functions are unclassified. The optimal size of the training set for practical use is subject to future research, preliminary results can be found in Section IV.
(3) Based on the training set, the tool automatically proposes classifications for each unclassified function pair, indicating whether two functions are equivalent and a number in [0,1] quantifying the certainty of this classification. In the example, the tool will predict that the two implementations are equivalent with high confidence.
(4) The analyst reviews the proposed classifications focusing on pairs that have a high confidence in being different as they usually have defects. The analyst reviews the suggestions using the tool where relevant differences are highlighted and irrelevant ones are faded out. The analyst documents detected defects. In the example, the tool fades out the differences (you can only see them from the very light-red background), and the analyst concludes that the differences are harmless and confirms that the functions are equivalent.
(5) Each function pair that the analyst classifies is incrementally added to the training set, which leads to reclassification of the unclassified function pairs. Consequently, the tool makes more appropriate suggestions over time. The analyst continues until all pairs are classified.

The tool contains four components: A *static source code analyzer* automatically detects differences in a function pair. A *machine learning model* automatically classifies function pairs as equivalent or different. This is done based on other function pairs that have similar differences and which already have been classified by an analyst. A *filtering mechanism* determines which differences are relevant and a *visualizer* presents the function pair, the filtered, automatically detected differences and the automatically conducted classifications to a human domain expert for analysis.

A. The Static Source Code Analyzer

Two implementations of a function, i_1 and i_2 (together they form the function pair), serve as input to the static source code analyzer, which calculates two outputs: (a) A set of differences in the source code, for example a variable or a function call which appears only in implementation i_1, but not in implementation i_2 or vice versa. We store these differences as strings and refer to each string as one *hint*. (b) One number quantifying these differences, for instance the number of differences found in (a). We call this number a *metric*.

The static analysis is performed in the following steps:

1) The static code analyzer splits files into functions.
2) To remove influence from coding style, the tool *uncrustify*[1] formats the code in a standardized way.
3) The functions are translated into our analysis framework using *srcML* [6], which creates an XML representation of the code that is afterwards translated into source code models that are analyzed statically.
4) Function pairs are built by matching functions based on their signature.
5) Several so-called *(data) extractor*s detect differences of various types, e.g. variable writings, variable readings, function calls, structural difference in the control flow, and many more. Each data extractor is called on each function pair, creating one set of hints and one metric per function pair and data extractor.

We created 16 data extractors where each one returns one single metric and a set of hints per function pair. To find these extractors, we developed an informal model (a.k.a. *anatomy*) that captures ”where things possibly can go wrong in the implementations of a SAL”. For example, issues such as differences in preconditions, return codes, etc. were captured as part of our anatomy. This guided us in implementing the extractors for locating non-trivial bugs. As one example, the *constant usage extractor* will take a function pair and extract all constants that are used in each implementation. The constant usage extractor will then compare the two sets and report constants that are only used in one of the implementations (this is called the *symmetric difference*). The names of the constants from the symmetric difference form the set of hints; in this case, for the metric the number of differences was chosen. For example, if the constants `OS_SUCCESS` and `OS_FAILURE` appear in one implementation and the constants `OS_SUCCESS` and `OS_NAME_TOO_LONG` appear in the second implementation, the symmetric difference results in `OS_FAILURE` and `OS_NAME_TOO_LONG`. The metric for this difference is the size of the symmetric difference, which is 2 in this case.

All implementations of the SAL must by definition have exactly the same signature as the interface because otherwise they would not be replaceable. Any differences in the signature are detected by the compiler and/or the linker. We analyzed how a function can change between two implementations of the same signature to detect differences between implementations that are not detected by the compiler and the linker. As functions can be described through input, behavior and output, we divided the data extractors into four groups: input, internal behavior, function calls (or external behavior), and output. Furthermore two additional indicators provide information based on traditional code metrics, such as *lines of code*. See Table I for a list of all extractors used by the tool.

The input extractors search the source code for differences related to inputs to the function. Usually, this means that implementations access varying resources as inputs during their execution. For example, if implementations use different constants, this usually leads to functional differences.

The second group of extractors detects differences based on integer or string values, and the way operators are used.

The third group of extractors detects differences in function calls through which implementations may access different underlying libraries or systems. Function calls can be of three types: First, *global function calls* are calls to functions that are available in all implementations, such as calls from one OSAL function to another OSAL function. Second, *local function calls* are calls to functions that are only available to one implementation and not to others. For example, system calls to the underlying OS are local function calls. Lastly, *local global function calls* are calls to functions that are available to all wrapper implementations (therefore the name global) but might not be included in a certain implementation because it is optional. Examples of local global function calls are calls to the C standard library `libc`.

The last group of extractors detects differences that are related to the output of an implementation.

B. The Machine Learning Model

All metrics of all function pairs (a matrix of numbers of size $|\text{functionPairs}| \times |\text{extractors}|$) serve as an input to the machine learning model. All function pairs include those that have been previously classified and their classification (*equivalent* or *different*) as well as those not yet classified. The machine learning model calculates two outputs: (a) for each unclassified function pair a classification of either equivalent or different and the confidence that this classification is correct and (b) the set of extractors that are the most significant for determining functional equivalence on this training set.

For the output in (a) the machine learning model uses the k-nearest neighbors algorithm (kNN) [8], [9] to calculate a prediction and a confidence. For each unclassified function pair kNN looks through the training set and searches for ”similar” function pairs in the training set. Similarity is determined by the Euclidian distance of the metrics, thus, the closer the two metrics tuples are, the more similar they are. From this list of the training set, the k-nearest neighbors algorithm suggests that the function pair is equal to the majority class (equivalent or different) of the k most similar function pairs. Confidence is then calculated based on the distribution within the k nearest neighbors, e.g. if 75% of the function pairs were equivalent then the confidence is 75%.

The calculation of the most relevant data extractors in (b) is performed by determining the most important metrics for the classification in the training set. Hence, we use so-called forward feature selection heuristics, which reduce the set of metrics (i.e. the set of extractors) to a more optimized one. This is done by subsequently adding extractor-by-extractor until the performance of the model stagnates[2]. Details on forward feature selection can be found e.g. in [9]. The metrics that remain after the feature selection are created by the

[1] Further information can be found under http://uncrustify.sourceforge.net/

[2] We will explain how to measure the performance of a model in Section IV.

TABLE I
ALL DATA EXTRACTORS IN USE

Group	Extractor Name	Description/Rationale
Input	Global reading differences	Differences in variables that are used as input for this function
	Project constant usage	Differences in use of constants that are defined within the OSAL
	Constant usage	Differences in all constants that are used
Function calls	Global function calls differences	Differences in function calls within OSAL
	Local function calls differences	Differences in calls only available to one implementation
	Local global function calls differences	Differences in calls that are external, but available to all implementations, e.g. `libc`
Internal behavior	Integer value differences	Differences in integer values used in the functions
	String value differences	Differences in strings used in the functions
	Operator differences	Differences of all operators used in the functions (union of operator-related extractors).
	Comparison operator differences	Differences of $\{<=, >=, ==, !=, <, >\}$
	Arithmetic operator differences	Differences of $\{++, --, +, -, /, *, \hat{}, \%\}$
	Logical operator differences	Differences of $\{\&\&, \|\|, \&, \|, !\}$
	Code-complexity difference	The difference in code complexity usually indicates differences in control flow
Output	Parameter writing differences	Differences in manipulation of parameters
	Global writing differences	Differences in manipulation of global variables
	Return code differences	Differences in possible return codes
Indicators	Lines of code differences	The difference in the length of both implementations
	Editing distance	The Levenshtein difference [7] of the implementations, i.e. the number of characters one needs to change to get from one version to the other version of the code

most important extractors for determining whether or not two functions are equivalent.

C. The Filtering Mechanism

With the hints from the static analysis component and the knowledge about the most suitable data extractors from the machine learning component as an input, we filter these hints in order to get to a more reduced and accurate set of hints.

In order to do this, our filter uses a two-stage approach: the first filter is based on the extractor (the type of a hint), whereas the second filter is based on the concrete hints.

First, we keep only hints that are generated by extractors that the machine learning model determined to be relevant. That way we can keep only hints created by the best performing extractors. Second, we remove hints that were repeatedly wrong. In practice it turns out that certain differences often occur in pairs. For example, an implementation for a POSIX system might always make a certain POSIX system call, whereas an implementation for a VxWorks system makes a similar system call with a different name. Hence, we build pairs of hints that appear together, from which we maintain a black list of hint pairs that appear in at least 10 equivalent functions, e.g. function calls to `pthread_mutex_lock` and `rtems_semaphore_obtain` (see example).

D. The Visualizer

This component is a graphical user interface that displays the different function pairs and their classification given by the machine learning model. In this view the analyst can further select a function pair for deeper analysis. The function pair is then displayed and the hints from the filtering mechanism are marked within the source code accordingly. The analyst can then decide whether or not two functions are equivalent, which leads to recalculation of the machine learning model.

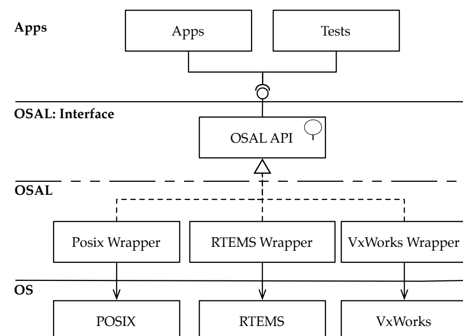

Fig. 4. The OSAL

IV. CASE STUDY

We studied NASA's OSAL because the CFS team was interested in understanding whether the approach would help them detect defects. We analyzed OSAL v3.2 which has 8.6 KLOC C code in about 100 functions, which are implemented for three different OSes. OSAL is implemented as a common interface with implementations of this interface for each supported OS. Each implementation wraps the underlying OS' system calls to carry out what the interface requires (see Fig. 4). Each implementation maintains global data structures where information about existing tasks, opened files etc. is stored. For each function defined in the interface, for example `OS_TaskExit`, there is one implementation for each supported OS, stored in separate files.

A. The Study

We analyzed all of OSAL's 100 functions that are present in all three implementations plus 4 internal functions which were present in two of the three implementations. This led to a total of 304 pairwise comparisons (function pairs).[3]

[3]If f_1 and f_2 are equivalent and f_1 and f_3 are equivalent, then via transitivity f_1 and f_3 must be equivalent, thus an opportunity for optimization.

Because the main goal of the approach is to detect inconsistencies related to functional equality, the evaluation focuses on understanding the potential of finding semantic differences of implementations. As the approach is semi-automatic we distinguish between two aspects of our analysis: First, we evaluate the approach's ability to detect defects. We do that by analyzing the defects we found in the OSAL case study and by providing a few examples. Second, we analyze the automatic classification part of the approach and measure to what extent the machine learning component provides useful classifications. We apply common machine learning evaluation methods to answer this question. To sum it up, we were interested in two research questions, the first one being:

RQ 1: Which defects can be found in OSAL when analyzing function pairs for functional equivalence?

In order to evaluate the usefulness of the approach, we studied whether issues stemming from differences in the OSAL code could be detected. In the 104 functions we found 84 issues in at least one of the three implementations. Of these, 57 were runtime issues, i.e. issues that may cause the system to behave in unexpected ways during execution. Since these issues were reported and confirmed as real issues by the OSAL team, 36 have been fixed while the remaining ones will be fixed in the near future. In total, 51 OSAL functions had issues. Please note that one function can reveal more than one issue as more than one problem might exist in an implementation. Table II shows the issues that were found in total.

TABLE II
OCCURENCES OF ISSUES IN OSAL

Type	#Issues
Precondition Checking Differences	13
Config Issues	9
Return Code Differences	24
Output Differences	18
Global Variable Writing Differences	15
Parameter Writing Differences	3
Parameter Checking	2

Precondition Checking Differences: Functions use preconditions to check that the values of input parameters meet certain conditions. We expect that all preconditions of equivalent functions are equivalent. Constant and operator extractors were used to detect 13 such issues. These issues could cause runtime problems and at this point 12 have been fixed.

Configuration Issues: Configuration constants are expected to be used in a consistent manner otherwise the OSAL may behave differently depending on which implementation is used. Constant extractors were used to detect 9 such issues. These issues would not cause runtime problems and at this point have not yet been fixed.

Return Code Differences: OSAL uses return codes for error handling. Consequently, all OSAL functions return a certain constant depending on the outcome of the function. An example is `OS_SUCCESS` for a successful operation and `OS_FAILURE` for a failing one. Return code extractors were used to detect 24 return code issues that were only used in one of the implementations. These issues could cause runtime problems. At this point 23 of them have been fixed.

Output Differences: Visual output includes, for example, messages written to standard output. The String extractor was used to detect 18 such issues. These issues would not cause runtime problems and at this point have not been fixed.

Global Writing Differences: OSAL defines a set of global variables and data structures that are used across all implementations and expect implementations to manipulate them in a consistent manner. The Global writing extractor was used to detect 15 such issues. These issues could cause runtime problems and at this point one has been fixed. The Global reading extractor did not result in any issues.

Parameter Writing Differences: When functions accept parameters by reference, they can return results by altering the parameters. Since this is a common way to return results from a function, we expect such parameter writing to be done in an equivalent way. The three detected issues could cause runtime problems, but at this point have not yet been fixed.

Parameter Checking: The functions implementing the SAL make use of the functions that the underlying (or wrapped) OS offers. Instead of risking to pass invalid parameter values, the wrapper checks that the values are valid before calling the underlying function. Differences between equivalent functions in how they do this checking may lead to defects. The constant extractor was used to detect two such issues. These issues could cause runtime problems, but at this point have not yet been fixed.

B. Examples of Defects

To provide a better understanding of what kind of issues can be found, we will discuss two different defects in detail[4].

Example 1: A Bug From Using a Wrong Constant: We found an issue where the initialization within `OS_API_Init` was using the wrong constant (see screenshot of our tool in Fig. 5). The implementation to the left wrongly initializes the array using the constant `OS_MAX_BIN_SEMAPHORES` while the implementation to the right correctly uses the constant `OS_MAX_COUNT_SEMAPHORES` to initialize the same array. This code only works correctly if both constants are initialized with the same value, which is a matter of configuration. However, flight software cannot take the risk that the code might work only by chance. This issue was detected because the constant extractor reported a difference. Our tool highlighted this difference in deep red since it determined that the issue was serious, while less important issues were colored light red. The OSAL developers confirmed and corrected the issue.

Example 2: A Bug from Inconsistent Side Effects: Our tool led us to the inconsistent manipulation of a variable in the VxWorks6 version of `OS_BinSemTimedWait` (an excerpt of the function is listed in Fig. 6, VxWorks6 left, RTEMS right). The responsibility of this function is to reserve a binary semaphore and provide a timeout for the

[4] See http://www4.in.tum.de/~femmer/works/femm12.pdf for more details.

```
/* Initialize Binary Semaphore Table */

for (i = 0; i < OS_MAX_BIN_SEMAPHORES; i++)
{
    OS_bin_sem_table[i].free  = TRUE;

    OS_bin_sem_table[i].creator  = UNINITIALIZED;
    strcpy(OS_bin_sem_table[i].name, "");
}

/* Initialize Counting Semaphores */
for (i = 0; i < OS_MAX_BIN_SEMAPHORES; i++)
{
    OS_count_sem_table[i].free    = TRUE;

    OS_count_sem_table[i].creator= UNINITIALIZED;
    strcpy(OS_count_sem_table[i].name, "");
}
```

```
/* Initialize Binary Semaphore Table */

for (i = 0; i < OS_MAX_BIN_SEMAPHORES; i++)
{
    OS_bin_sem_table[i].free  = TRUE;
    OS_bin_sem_table[i].id        = NULL;
    OS_bin_sem_table[i].creator  = UNINITIALIZED;
    strcpy(OS_bin_sem_table[i].name, "");
}

/* Initialize Counting Semaphore Table */
for (i = 0; i < OS_MAX_COUNT_SEMAPHORES; i++)
{
    OS_count_sem_table[i].free    = TRUE;
    OS_count_sem_table[i].id  = NULL;
    OS_count_sem_table[i].creator= UNINITIALIZED;
    strcpy(OS_count_sem_table[i].name, "");
}
```

Fig. 5. A bug from using a wrong constant (POSIX left, VxWorks6 right)

```
OS_bin_sem_table[sem_id].current_value--;
if (semTake(OS_bin_sem_table[sem_id].id, sys_ticks) != OK)

{
    OS_count_sem_table[sem_id].current_value++;
    return OS_SEM_TIMEOUT;

}

return OS_SUCCESS;
```

```
OS_bin_sem_table[sem_id].current_value--;

status =   rtems_semaphore_obtain(OS_bin_sem_table[sem_id].id, RTEM

switch (status)
{
case RTEMS_TIMEOUT:

    OS_bin_sem_table[sem_id].current_value++;
    status = OS_SEM_TIMEOUT;
    break;

case RTEMS_SUCCESSFUL:
    status = OS_SUCCESS;
    break;

default:

    OS_bin_sem_table[sem_id].current_value++;
    status = OS_SEM_FAILURE;
    break;
}
return status;
```

Fig. 6. A bug through inconsistent side effects (VxWorks6 left, RTEMS right)

creation. OSAL maintains a global data structure for all semaphores called `OS_bin_sem_table`. This structure contains a field called `current_value` that stores a value indicating how many units of this semaphore are available. It is properly decreased at the beginning of the two implementations. However, in an error case (for example, if the semaphore cannot be allocated as planned) we need to revert this decrement and free the semaphore again, which is performed at the end of the function. Looking at the VxWorks6 implementation more closely, we can see that it increases the counter of the `OS_count_sem_table` array, instead of the `OS_bin_sem_table` array, thereby corrupting both arrays. This difference was detected by the Global writing extractor. There are also other issues in this implementation, but they are not in the focus of this example. The bug was reported to the OSAL team, who confirmed and corrected the issue.

RQ 2: To what extent can the machine learning model classify functional equivalence in OSAL?

To understand the performance of the automatic part of the approach, we evaluated how good the classifier is. In information retrieval this is usually analyzed using three performance metrics: *precision*, *recall* and *accuracy*. The precision value calculates the quality of the classifications, whereas the recall describes its completeness. As such they are conflicting goals: a higher precision usually results in a lower recall and vice-versa. Both precision and recall are calculated for a certain *positive class*, hence describing the performance when looking for equivalent (EQ) or differing (DIFF) function pairs. For example, when we characterize equivalent function pairs as the positive class, the true positives are those function pairs that have been correctly classified as equivalent. If we characterize different function pairs as the positive class, the true positives are those function pairs that have been correctly classified as different. Accuracy only calculates the percentage of the correct guesses (irrespective to the positive class). These performance metrics are calculated by using the classifier's true positive (tp), true negative (tn), false positive (fp) and false negative (fn) classifications:

$$\text{Prec} = \frac{tp}{tp + fp} \quad \text{Rec} = \frac{tp}{tp + fn} \quad \text{Acc} = \frac{tp + tn}{tp + fp + tn + fn}$$

The choice of the training set has a strong influence on the performance of the classifier. In order to minimize this effect and to ensure representativeness of the evaluation, two strategies are usually applied in machine learning: first, *n-fold cross-validation* splits up the set of classified function pairs in n subsets. Afterwards $n - 1$ subsets are used to train the machine learning model, and one subset is classified. From this classification we count the number of correct suggestions of the machine learning model. This is repeated n times until each subset has been classified. Three-fold cross-validation is

a widely used approach [9]. Second, *repetition* prevents that the choice of how to split up the n subsets has an impact on the final result. Repetition means that the analysis is replicated i times and the results are averaged using the arithmetic mean. We used repetition ($i = 100$) and three-fold cross-validation to evaluate the quality of the classifier. Data was analyzed with *R* and *RapidMiner*. We defined the golden standard, which was used for training, through thorough manual inspection.

The last ingredient to understand the performance of the machine learning classifier is the comparison of the performance metrics against alternative approaches. We created three different baseline classifiers: (1) In machine learning it is common to evaluate the performance against a baseline classifier *ZeroR*, which is a naive approach that takes into account the bias of the data. To do so, ZeroR always suggests the majority of classifications, i.e. in our case there are more equivalent than different function pairs, hence, ZeroR always suggests that function pairs are equivalent. ZeroR is useful as a baseline in cases where no previous data has been collected, like in this case. (2) In addition we wanted to compare the approach to traditional diffing, such as the UNIX diff command, as it is the most common tool for programmers to solve the problem. We used the DiffPlex[5] library, which is a character-by-character comparison of the source code. Consequently, two functions are different if and only if their implementations differ in at least one character. (3) We improved this library to reduce the impact of different code styles using the *uncrustify* tool, resulting in identical formatting. This extended DiffPlex classifier suggests that two functions are different if and only if the formatted versions of their code bases differ. All performance metrics were calculated for all classifiers.

Table III shows the results of the evaluation of our classifier. The performance metrics are visualized in rows, the columns show the different classifiers we used for comparison.

TABLE III
THE PERFORMANCE OF THE CLASSIFIER FOR OSAL

	Classifier	ZeroR	DiffPlex	Extended DiffPlex
Accuracy	88.8%	59.9%	44.7%	54.6%
Precision (EQ)	88,9%	59,9%	100,0%	100,0%
Precision (DIFF)	88,5%	0%	42,1%	46,9%
Recall (EQ)	92,8%	100%	7,7%	24,2%
Recall (DIFF)	82,8%	0%	100%	100%

When comparing the classifier's performance to the alternatives we can see that the classifier does not give perfect results but that it is a very high-level compromise between precision (i.e. minimization of wrong suggestions) and recall (i.e. optimizing for completeness). The average accuracy is 89% with a variance of 1.25%. This means that we can build a classifier for OSAL, which, if trained with $2/3$ of the examples, can classify unknown function pairs correctly in about 89% of the cases. This is significantly better than ZeroR or the DiffPlex classifications. In fact, both DiffPlex approaches perform even worse than ZeroR classification.

[5]http://diffplex.codeplex.com

To understand in which cases the classifications between the approaches vary, we have to understand that the DiffPlex approaches check which implementations are syntactically equal (identical source code, only formatting differences); hence, they classify everything as different that is not identical. Thus, the implementations that really differ are a subset of those that the DiffPlex approach suggests to be different. This explains the 100% recall of the DiffPlex approaches for the class "different". Yet, the precision is low (42%) meaning that only a small subset of the functions that differ in source code is really semantically different. This precision can be improved marginally (47%) by the extended DiffPlex approach.

However, our approach is not perfect either. The fact that the false positives and negatives still form 11% of our classifier's suggestions shows that reviewing is still necessary. This supports creating a tool that includes machine learning support, but where the analyst still reviews the proposed classifications.

C. Lessons Learned from the Case Study

Through this study we learned the following lessons. For the developer, first creating an anatomy of possible SAL issues was helpful. Without an anatomy we would not have been able to implement appropriate data extractors and analyzers for our tool. Another observation is that the classification improved after more and more functions were classified. This indicates that the approach would pay off as soon as we repeat the analysis with newer versions of the system.

For the analyst who used the tool, the machine learning model was very useful because it automatically detected function pairs that are equivalent even though they were slightly different. It did so by quickly learning that certain differences are irrelevant to functional equivalence, i.e. if everything is equal except for function calls to the OS. Similarly, the machine learning model was often also helpful as a filter. However, it was not always accurate and at times filtered out the important parts. It was difficult for the analyst to understand the differences for function pairs where the implementations were very different. The reason for this is that the tool assumes that the function pairs are somewhat similar in structure and when this is not the case, it can only provide limited support. In addition, the analyst felt that a huge benefit came from the ability to view the whole API, including all implementations, next to each other. This was especially useful since the analyst could quickly jump into any comparison.

D. Threats to Validity

a) Threats to Internal Validity: Threats to internal validity are related to whether the results stem from the methodology under study and not from other facts and circumstances. We addressed the threat that the golden standard might not be accurate by thorough manual inspection of the source code as well as letting the CFS team review the issues and confirm that they were significant. This is why we assume that most differences have been detected and are represented in the golden standard. Furthermore, excessive automated testing

(approx. 150 tests) of the developed tool addressed threats to internal validity through bugs.

b) Threats to External Validity: Threats to external validity question the ability to generalize the results to other software systems. The machine learning approach could be subject to over-fitting, i.e. it is specialized on OSAL, but performs poorly for other cases. We applied cross-validation and repetition to address this threat for all statistics within this study. Hence, the different possible choices for a training set are addressed by well-known techniques from the field of machine learning. However, it remains unclear whether the success in this case study depends on the characteristics of the OSAL system or on other circumstances not accounted for. For example, OSAL uses well defined return codes, while other systems might not. Hence, the classifier might not work as well on such systems. Lastly, even though the approach was implemented and evaluated for C code, there is nothing in the approach that prevents one from applying it to systems written in other languages, as long as one can parse the abstract syntax tree of the code. Yet, some amount of work might be needed to adapt the extractors to the semantics of another language, for example, translate how to detect which global variables are written to and read from.

V. Discussion

A Note on the Bugs Detected in the Context of OSAL: The following section was provided by one of the CFS developers in response to the results reported in this paper. While all detected issues have been confirmed by the OSAL team, the context for the OSAL development might explain why there are defects in the first place. The OSAL was not a top-down managed effort and the development process did not state "let's build a product line for platforms X, Y, and Z and do a normal project development with a staff, budget, and schedule." Instead, the OSAL is the result of a bottom-up effort and each platform abstraction was added as it was needed. For example, missions such as SDO and LRO are both VxWorks-based so since the OSAL works in a reliable way with these platforms these two projects are okay. The defects that the missions found in the process of testing their mission software, which includes their internalized version of the OSAL source code, were not necessarily reported back to the OSAL team. In addition, these missions only use a limited set of the OSAL functions. Thus there are OSAL functions that have been added to make the OSAL more general and applicable to more missions. Thus these functions are currently not used by a mission, and since they are not used they have not been extensively tested. Nevertheless, one of the problems with bottom-up development (due to project-only-funding) is that it is hard to develop and maintain a comprehensive test suite that covers all OSAL functions for all supported OSes including the functions that are currently not used by any mission. The reality of this bottom-up piecemeal approach supports the call for an approach that detects defects in wrappers.

A Note on the Applicability of the Approach: It is important to note that the approach is based on the assumption that different implementations share several traits, such as constants used, global functions called, error codes returned, etc., and seems to work well in that context. However, this assumption may not hold for all systems. For the NASA case study this assumption holds and the approach is applicable. However, we cannot automatically take this assumption for granted when moving to another context where the implementations might be largely unrelated but still functionally equivalent, e.g. if the implementations are based on largely different data and function call structures. Furthermore, the approach is designed to detect differences between two implementations. A bug that is present in both implementation or bugs that are based on certain semantics, e.g. dead code or similar, will not be found by the approach. It is also important to note that the approach at this point is neither complete nor sound, meaning that not all issues are detected (i.e. there are false negatives) and that some differences may be wrongly reported as issues (i.e. there are also false positives). Yet, our main insight was that even for a very difficult problem a simple approach can be of great help in certain circumstances.

VI. Related Work

Determining whether two programs are equivalent reaches back into the 1950ies. The earliest paper we found was by Hoare [10] who references papers discussing the topic in 1958.

A. Formal Verification

Recently, equivalence of components came back into focus with applications in regression analysis, e.g. [11] analyzes equality of different representations of code for embedded systems. Also, verification of abstract representations of OSes against their implementations is possible, but expensive [12]. We constructed a cheaper, but less rigorous approach.

The SymDiff (Symbolic Differencing) project focuses on regression analysis and extracts information displaying the semantic differences between versions to detect undesired side effects. In [13] the authors define conditional equivalence as two functions being semantically equivalent under certain inputs and implement a version for the intermediate language BoogiePL. This is extended for C in [14]. In [15] the authors automatically extract termination conditions and summaries from source code using laboratory examples. [16] uses forward symbolic execution based on KLEE [17] to analyze standard libraries for differences. In [3] and [18] differential symbolic execution is used for regression analysis, which extracts only differing parts and performs symbolic execution on these to find differences. Similar approaches are presented in [19].

These works address problems that are similar to the ones we were facing with equivalence of SALs. However, they make assumptions that we cannot make. Existing approaches often require the source code of the underlying systems or at least a stub implementation. Unfortunately, source code is not always available. Development of meaningful stubs is effort intensive. For other approaches (such as [20] or [21]) the code must be executed, which is not possible in OSAL, for both licensing and application domain reasons (space software).

B. Diffing Approaches and Program Matching

[22] uses textual diffing for regression analysis where enhanced Control Flow Graphs (CFGs) are used to identify semantic changes. Other early work on semantic diffing is described in [23], as well as an algorithm that identifies changes in CFGs. However, changes in CFGs are only a small part of possible semantic-preserving changes. We search for differences in CFGs as well, but we also detect return code issues and differences in configuration constants.

[24] summarizes the state of the art in the field of program element matching where parts of source code are matched against each other. They evaluate different methods based on hypothetical change scenarios. Their goal is to support the development of software that origins as clones. They try to determine whether a bug impacts more than one piece of code while we try to locate bugs using differences in source code and train a machine to automatically detect similar bugs.

[25] analyses different versions of VHDL code. They enhance a regular diff-tool based on the semantics of VHDL, thereby improving the precision and recall of the diff tool. [26] creates a comparable approach for Java. This is similar to the creation of our extended DiffPlex, which we used as a baseline classifier in the evaluation. However, we are using the semantics of ANSI C.

We apply machine learning to filter the results of the static analysis approach. This idea has been successfully applied before (e.g. in [27]). However, to the best of our knowledge, this is the first attempt to apply machine learning for equivalence analysis. In contrary to these works, our approach not only detects differences based on the semantics of C, but combines these differences with source code measures to train a machine learner to prioritize high-risk source code differences. Furthermore, we embed the analyst's judgment into the process of prioritizing differences.

VII. Conclusion and Future Work

In this paper, we first introduced a simple approach based on tool-supported reviews leveraging source code measures and machine learning to detect inconsistencies in software abstraction layers. Second, we studied the use of OSAL, a high-quality reusable framework for space missions that was developed by the NASA CFS team. Our case study especially focused on defect detection strategies related to OSALs and how non-equivalent wrapper implementations can be detected. The main question was whether our approach would serve as a complementary verification method for the CFS team. We evaluated the approach by analyzing NASA's OSAL that has more than 100 interface functions and is implemented for three different OSes (RTEMS, POSIX, and VxWorks). The approach detected 84 corner-case issues of which 57 turned out to be bugs that could have resulted in runtime failures.

Since the approach turned out to be useful, we will analyze whether the different implementations of the NASA core flight executive layer (cFE) are equivalent.

Acknowledgment

This work is supported by NASA's Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) ”Architectural Analysis of Dynamically Reconfigurable Systems” project. We especially thank Lisa P. Montgomery, Alan Cudmore, and Art Hughes for their support of our work.

References

[1] W. Chu, W. Li, T. Mo, and Z. Wu, “A Context-Source Abstraction Layer for Context-aware Middleware,” in *ITNG '11*.
[2] R. S. Oliver, I. Shcherbakov, and G. Fohler, “An operating system abstraction layer for portable applications in wireless sensor networks,” in *SAC '10*.
[3] S. Person, M. Dwyer, S. Elbaum, and C. S. Pasareanu, “Differential symbolic execution,” in *FSE '08*.
[4] H. Chen, D. Dean, and D. Wagner, “Model checking one million lines of C code,” in *NDSS '04*.
[5] D. Ganesan, M. Lindvall, D. McComas, S. Slegel, and B. Medina, “Architecture-based Unit Testing of the Flight software Product Line,” *SPLC '10*.
[6] J. Maletic and M. Collard, “Supporting source code difference analysis,” in *ICSM '04*.
[7] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics Doklady*, vol. 10, no. 8, 1966.
[8] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE Trans. on Information Theory*, vol. 13, no. 1, 1967.
[9] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. Morgan Kaufmann, 2011.
[10] C. Hoare, “An axiomatic basis for computer programming,” *Comm. of the ACM*, vol. 12, no. 10, 1969.
[11] X. Feng and A. Hu, “Cutpoints for formal equivalence verification of embedded software,” in *EMSOFT '05*.
[12] G. Klein et al., “seL4: formal verification of an operating-system kernel,” *Comm. of the ACM*, 2010.
[13] M. Kawaguchi, S. Lahiri, and H. Rebelo, “Conditional Equivalence,” Microsoft Research, Tech. Rep., 2010.
[14] S. Lahiri and S. Qadeer, “Back to the future: revisiting precise program verification using SMT solvers,” *ACM SIGPLAN Notices*, 2008.
[15] C. Hawblitzel, M. Kawaguchi, S. Lahiri, and H. Rebelo, “Mutual Summaries and Relative Termination,” Microsoft Research, Tech. Rep., 2011.
[16] D. Ramos and D. Engler, “Practical, Low-Effort Equivalence Verification of Real Code,” in *CAV '11*.
[17] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *OSDI '08*.
[18] S. Person, G. Yang, N. Rungta, and S. Khurshid, “Directed incremental symbolic execution,” *ACM SIGPLAN Notices*, vol. 47, no. 6, 2012.
[19] T. Sakunkonchak, T. Matsumoto, H. Saito, S. Komatsu, and M. Fujita, “Equivalence checking in c-based system-level design by sequentializing concurrent behaviors,” in *ACST '07*.
[20] K. J. Hoffman, P. Eugster, and S. Jagannathan, “Semantics-aware trace analysis,” in *PLDI '09*.
[21] M. Vouk, “On Back-to-Back Testing,” *Information and Software Technology*, vol. 32, no. 1, 1990.
[22] S. Horwitz, “Identifying the semantic and textual differences between two versions of a program,” University of Wisconsin, Tech. Rep. 6, 1989.
[23] J. Laski and W. Szermer, “Identification of program modifications and its applications in software maintenance,” in *ICSM '92*.
[24] M. Kim and D. Notkin, “Program Element Matching for Multi-Version Program Analyses,” in *MSR '06*.
[25] A. Duley, C. Spandikow, and M. Kim, “Vdiff: a program differencing algorithm for Verilog hardware description language,” *ASE '12*.
[26] T. Apiwattanapong, A. Orso, and M. J. Harrold, “JDiff: A differencing technique and tool for object-oriented programs,” *ASE '06*, vol. 14, no. 1.
[27] E. Bodden, P. Lam, and L. Hendren, “Finding programming errors earlier by evaluating runtime monitors ahead-of-time,” in *FSE '08*.