VALID 2018

The Tenth International Conference on Advances in System Testing and Validation Lifecycle

ISBN: 978-1-61208-671-2

October 14 - 18, 2018

Nice, France

VALID 2018 Editors

Jos van Rooyen, Identify - Software Quality Services, the Netherlands

VALID 2018

Forward

The Tenth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2018), held on October 14 - 18, 2018- Nice, France, continued a series of events focusing on designing robust components and systems with testability for various features of behavior and interconnection.

Complex distributed systems with heterogeneous interconnections operating at different speeds and based on various nano- and micro-technologies raise serious problems of testing, diagnosing, and debugging. Despite current solutions, virtualization and abstraction for large scale systems provide less visibility for vulnerability discovery and resolution, and make testing tedious, sometimes unsuccessful, if not properly thought from the design phase.

The conference on advances in system testing and validation considered the concepts, methodologies, and solutions dealing with designing robust and available systems. Its target covered aspects related to debugging and defects, vulnerability discovery, diagnosis, and testing.

The conference provided a forum where researchers were able to present recent research results and new research problems and directions related to them. The conference sought contributions presenting novel result and future research in all aspects of robust design methodologies, vulnerability discovery and resolution, diagnosis, debugging, and testing.

We welcomed technical papers presenting research and practical results, position papers addressing the pros and cons of specific proposals, such as those being discussed in the standard forums or in industry consortiums, survey papers addressing the key problems and solutions on any of the above topics, short papers on work in progress, and panel proposals.

We take here the opportunity to warmly thank all the members of the VALID 2018 technical program committee as well as the numerous reviewers. The creation of such a broad and high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and efforts to contribute to VALID 2018. We truly believe that thanks to all these efforts, the final conference program consists of top quality contributions.

This event could also not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the VALID 2018 organizing committee for their help in handling the logistics and for their work that is making this professional meeting a success. We gratefully appreciate to the technical program committee co-chairs that contributed to identify the appropriate groups to submit contributions.

We hope the VALID 2018 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in system testing and validation. We also hope Nice provided a pleasant environment during the conference and everyone saved some time for exploring this beautiful city.

VALID 2018 Steering Committee

Andrea Baruzzo, IDS Interaction Design Solutions, Italy
Tadashi Dohi, Hiroshima University, Japan

VALID 2018

Committee

Table of Contents

System Debug and Validation: Use case Based Perspective

Bhushan Naware
MIG, Intel Technologies India Pvt Limited
Bangalore 560103 India
Email: bhushan.g.naware@intel.com

Arun A Pai
WSS, Intel Technologies India Pvt Limited
Bangalore 560103 India
Email: arun.a.pai@intel.com

Ravinder Singh
WSS, Intel Technologies India Pvt Limited
Bangalore 560103 India
Email: ravinder.m.singh@intel.com

***Abstract* – Concept and Design of systems with sheer complexity at various abstraction levels is becoming tedious and time consuming process. To comply with the expected requirements, subsequent validation and verification becomes even more time consuming and expensive. When it comes to platform level validation and debug, there are various fronts that are to be looked at with great depth. In case of laptops/desktops the system stack includes hardware, silicon, firmware, bios, operating system, various drivers and applications. In complex systems, finding root cause of issues caught at platform validation is challenging and increases debug throughput. In this paper, we will introduce a methodology for validation and debug that could be applied across similar systems. This methodology is bound to shorten the life span of test plan creation, early identification, debug and root cause of issues. This will result in cost saving and shorter time to market.**

Keywords: Test Plan; Use case; Scenario; Win-DBG; JTAG; Silicon Debug.

I. INTRODUCTION

Every year the computing systems are becoming more complex and as a result there is an increase in overall product life cycle time starting from concept, design, development and validation. The validation and platform debug needs to be very efficient and test cases needs to be derived from real use cases in-order to exercise all corner scenarios. The other possibilities can also include stress testing of the systems that has to be done with existing and newer features. Stress and Stability testing consumes the maximum duration of validation as the test duration spans across days, these test increases the workload of the platform validation teams exponentially due to sporadic failures which takes more time to repro. The methodology that is proposed in this paper can be used by extensive number of teams/customers that are working on platform validation; be it original design manufacturer or original equipment manufacturer or the in-house validation teams that are responsible for product readiness and deployment. This paper provides an introduction to the concept of use cases as one of the obelisk of validation metric in order to scale the validation and also make the entire coverage robust and more adaptive. Using the concept of use cases to bolster the system validation, there is a preeminent advantage in the issue debugging and also gauging of coverage across platform which can provide status for overall product readiness with respect to the quality requirements.

Currently the state of the art validation methodologies that are used by original equipment manufacturers and original device manufacturers and also the in-house validation teams is based on feature based approaches and the one proposed here currently is being used for the first time in broad system level context.

The paper is outlined as follows. In section II, the concept of use cases is explained. In section III the generic approach of platform testing and debug via use cases is proposed. In section IV with one of the running examples the concept of use case and usage revelation is brought forward, also ease of debugging of issue is explained in section V. The paper finishes with conclusion in section VI.

II. HYPOTHESIS

The conventional way in which the system validation is performed revolves around the new feature debut, in a particular platform whether it is a hardware or software, then checking if the standalone operation of feature matrix is proper, and if the answer to that is yes; then subsequently it has to be validated and substantiated for the different flows at the platform level. Considering there is sprouting feature list and also the new evolutions of system use patterns; platform test plan intricacy & validation cycle time increases multifold.

In order to mitigate and get the details on the above list of features sorted out, there is a need of mechanism that would give us portable and more systematic way of tracking features at platform, which would eventually touch all the underlying sub-features. Hence, instead of looking at the system from new features standpoint we look at the system from the usage scenarios.

The end-user when aims to use the system, what is the way in which the system is used. Complexity of such flows is taken into account and once that use case list is in place, we have more or less a constant feature list that would cover a particular domain. Once the list of use cases is fairly constant platform over platform there is more transparency in terms of tracking. Further to that depending on a new feature introduction, be it architectural or any software dependent it can be knit into the particular use case; solving both the purposes, keeping the main test tracking list constant and also incorporating the new testing scenarios. We are following the inverted edifice approach to solve the test case intricacy and other allied issues.

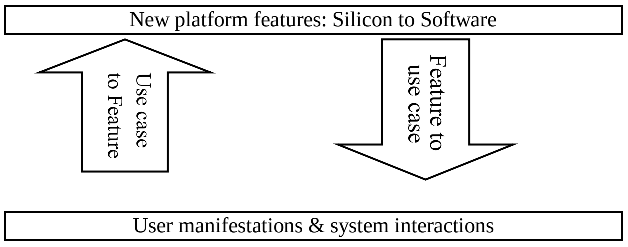

New platform features: Silicon to Software
Use case to Feature
Feature to use case
User manifestations & system interactions

Figure. 1 Inverted edifice: use case vs feature based approach.

Right hand side of Figure 1 represents the Conventional mode while the left side represent the proposed methodology. As paper progresses, it would be more relevant and clear how can we have ease of validation and debug addressed by new use case based approach.

III. TEST AND DEBUG TRIGGERED VIA USE CASE

Having understood the use case approach, we need to check the details on the usage of the same with respect to the validation and debug on real world platform. The following section takes a look at both the validation and debug of platform spanning out with the use case based approach from an idealist system standpoint. In-order to understand about applicability of use case approach for debug and validation strategies, we need to first observe on type of abstraction layers we have and then pertaining to abstraction layers, what kind of validation problems and debugging complexities can precede. Identification of problem, with anticipated complexities & trying to address it with proposed approach would help, in reduction of both the validation as well as debugging related issues. This is seen in the following section with hibernate system state example.

Taking into account the "outside in approach" we normally see what a silicon (CPU) offers, after that we design the features, as well as other supported customaries. When we have the requisite hardware, i.e., silicon and board, it comes to the BIOS, Firmware's & device drivers. When all these things are good, we then move towards the choice of "OS" and the subsequent test requirements that are needed in-order to perform our validation. From the representation in Figure 2, it becomes much evident on what complex level the System validation happens.

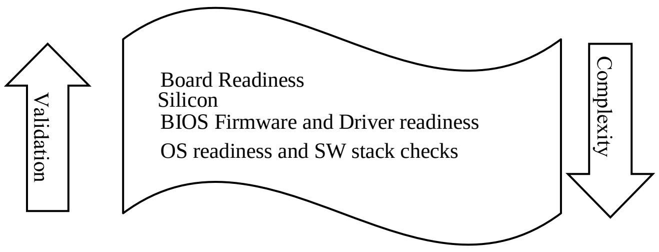

Validation
Board Readiness
Silicon
BIOS Firmware and Driver readiness
OS readiness and SW stack checks
Complexity

Figure. 2 Silicon to Software features depiction and traditional testing methods

Validating a scenario with proper use case defined becomes easier to test & articulate. Let us take a look at small example to explain how a use case definition can help to ease the complexity of validation. Goal is to validate system states that a system under test supports, and see what the test coverage is attached to the particular use case, map it back and get information about supported power states. Using various tools we can get re-confirmation that indeed these are the power features that we are expecting on the system. Once the existing use case is available amalgamating any new power feature onto the system power state matrix, becomes quite a simple task. Figure 3 provides an insight into one of the systems and the various power states that it supports in particular. This is a toned down version of multiple states via which the system can navigate through in different phases of validation and actual usage.

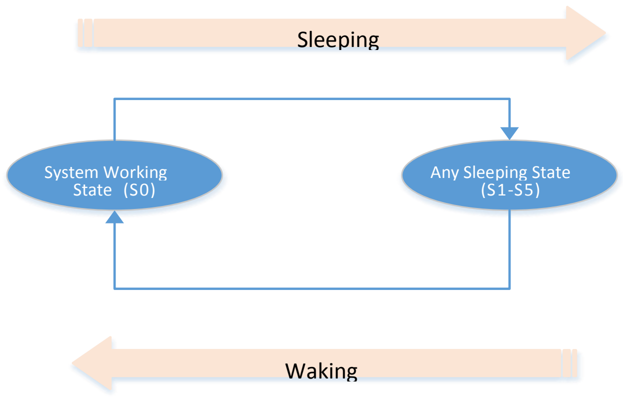
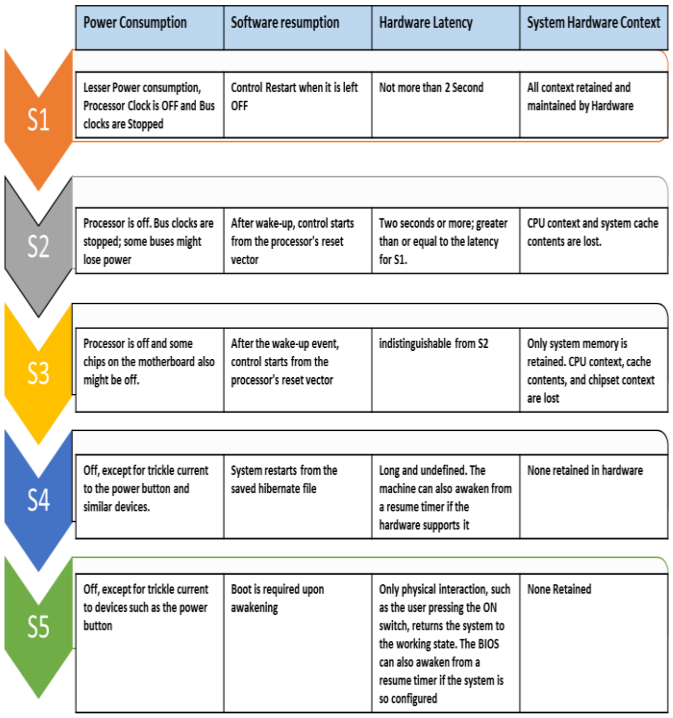


Figure. 3 System and corresponding supported and un-supported power states.

In addition to the validation strategies of available power states & checking on the coverage gap of existing power states or completely abstaining power state, we can also look at the use case definition as one of the major pillars for debug.

Consider one of the debugging scenarios here with platform power policy and the way failure condition is debugged. We expect S4 (Hibernate) state as a default platform state present, but for some reason we have the S4 (Hibernate) not mentioned in the available states list. Then, from the use case lists we can take starting point as absence of S4 (Hibernate) state & what are the platform use cases that would be hindered and in-turn what feature lists cannot be tested. Then from Figure 2. Silicon to Software depiction, we can check that what can be the probable cause of system state failure, whether it maps to silicon abstraction layer or due to any other failure. Once the leads are generated the debug process direction is decided accordingly. Say we have issue with the OS then follow up software debugging is done and subsequent resolutions would get us the issues sorted out.



	Power Consumption	Software resumption	Hardware Latency	System Hardware Context
S1	Lesser Power consumption, Processor Clock is OFF and Bus clocks are Stopped	Control Restart when it is left OFF	Not more than 2 Second	All context retained and maintained by Hardware
S2	Processor is off. Bus clocks are stopped; some buses might lose power	After wake-up, control starts from the processor's reset vector	Two seconds or more; greater than or equal to the latency for S1.	CPU context and system cache contents are lost.
S3	Processor is off and some chips on the motherboard also might be off.	After the wake-up event, control starts from the processor's reset vector	indistinguishable from S2	Only system memory is retained. CPU context, cache contents, and chipset context are lost
S4	Off, except for trickle current to the power button and similar devices.	System restarts from the saved hibernate file	Long and undefined. The machine can also awaken from a resume timer if the hardware supports it	None retained in hardware
S5	Off, except for trickle current to devices such as the power button	Boot is required upon awakening	Only physical interaction, such as the user pressing the ON switch, returns the system to the working state. The BIOS can also awaken from a resume timer if the system is so configured	None Retained

Figure. 4 System power state details

IV. VALID USE CASE ANALYSIS

This section demonstrates the usage of this model to get more robust understanding of use case application. For explanation purpose, system state Hibernate a.k.a. S4 is considered. Validation plans & tests are derived from use case scenarios. Various Combination and tests are planned with S4 entry/exit criteria kept in mind. Coverage is quantified with features planned and validation matrix created subsequently for an end to end use case and flow.

Basic understanding of Hibernate use case, provides information such as the condition of system, power consumption during hibernate, input wake mechanism, how system should behave after wake and what to restore after hibernate exit. Additionally various user scenarios which include multi domain interactions are also covered. Table I enlists some of the features/test that needs to be checked and covered during Hibernate use case validation.

TABLE I. FEATURE/TEST CASE CHECKS DURING HIBERNATE USE CASE

SI No	Condition in Hibernate	
	Scenario	When
1	System in off condition	Yes during entry
2	Power consumption status	While in S4 system should measure the lowest power
3	Wake scenario	Wake using USB, LAN or any other input source based on the system feature.
4	Context Saving	While entry, hiber file should be generated with all the active context stored and on exit it should retrieve all the context from the Hiberfile.sys
5	Battery Management	System should trigger Hibernate based on the amount of time the system is in idle.
6	Responsiveness	Involves Time taken for entry and exit for hibernation
7	Memory Management	Check System memory Decomposition when resumed from Hibernation
8	Video/Audio Resume after Hibernation	Context shouldn't be lost and user should be able to resume the MM content
9	Input Sequence	What type of input sequence need to be planned for entry such as power button, via OS , using scripts etc.

Understanding few of the scenarios would help in gauging usage of this model on real system cases. Gradually starting with the functional test then moving to inter operational tests, stress test and finally to the reliability checkouts is the methodology of this use case model.

In every stage of checkouts, various tests are performed and result is measured against expected outcome. Starting with functional checks where the basic entry/exit of hibernation is tested and expectation is to have all the precondition met. Entry to S4 when initiated, triggers following processes all apps drivers and services are notified, all the system context is saved on the boot media. Resumption from S4 is determined by OS boot manager by detecting a valid hibernation file, after that it directs the system to resume, restoring the contents of memory and all architectural registers, the contents of the system memory are read back in from the disk, decompressed, and restored back to its original state [4].

After functional tests, few inter-operability scenarios are run to make sure system memory is checked properly with all active context saved and resumed. The example scenarios for system memory check would be video player run resuming from where it was paused or YouTube streaming window reloaded and paused, etc. Consecutive Hibernate cycles, with counts gradually increasing from 100, 500 to 1000 are checked. This provides the overall stability and confidence on the system reliability. Additionally inclusion of traffic along with hibernate cycle is done where, scenarios such as Bluetooth file transfer and S4 cycles simultaneously, are run for multiple iterations.

TABLE II DIFFERENT HIBERNATION FILE BASED ON FAST STARTUP OPTION

Hibernation File Type	Default Size	Supports
FULL	40% of physical memory	Hibernate, Hybrid Sleep, fast Startup
REDUCED	20% of physical memory	Fast Startup

V. ISSUE DEBUG ON HIBERNATION

Debugging any issue from platform remains challenging and most troublesome due to the sheer number of variables that can affect the flow. It becomes more tedious and cumbersome process if any power flow is involved as there are numerous transitions which increases failure probability & debug complexities. Of the all power state flow Debug of Hibernation use case remains is one of the toughest.

For any power flow it has two contexts, 1st being entry and 2nd being exit or resume from concerned power state. Issues mostly prevail among these two context. The issues seen can be categorized into following failure buckets.

1. Context not getting saved after resuming from S4
   a. The system is not taking the S4 flow and entering into other alternative Power flow path such as S3 or S5.

2. Devices not recognized after resuming from S4
   a. Multiple devices gets lost or doesn't detect after resume such as storage devices like USB, Yellow bangs to various modules such as Connectivity modules, IO or any controllers.

3. Soft Hangs
   a. Recoverable Hangs which can be due to issues in device driver loading after resume.
   b. Unrecoverable Hangs or device lost while resuming resulting in Memory dump such as Blue Screen of Death (BSOD), Green Screen of Death (GSOD).

4. Hard Hangs
   a. Non Recoverable error or hang observed resulting in system not responding towards any of the user commands
   b. These issues can be due to IP hangs or Silicon hangs. Debugging done via Joint Test Action Group (JTAG).

5. Responsiveness
   a. Time taken to enter the Hibernate is more compare to the Target specified.
   b. Time taken to resume from Hibernate fails to meet the target specified.

6. Auto Wake Issues
   a. Systems wake as soon as it enters S4.
   b. System wakes from S4 after certain duration.

All failure needs a different debug approach, in order to achieve the best results. Each failure above needs dedicated effort and support to root cause and narrow down upon the exact problem causing component. At high level we can understand the debug strategy using following flow chart given in Figure 5 and aligning it with the use case based model gives us the flexibility of getting things done at much faster, organized and streamlined way. We start the debug to check if it is a hard hang and if it is the case we need to use hardware mechanisms and tool like JTAG to scan inside the silicon using its Design for Test (DFT) capabilities. If there are only soft hung seen, then we need to get it bucketed in sub-category and pursue a different method of debug regarding the same. This flow chart explains at a very high level of abstraction of a well knitted and branched out debug tree. With more and more defects the tree would fan out to utmost complexities and that is where the use case scenarios will come in handy to identify the feature dependencies and debug them as applicable.

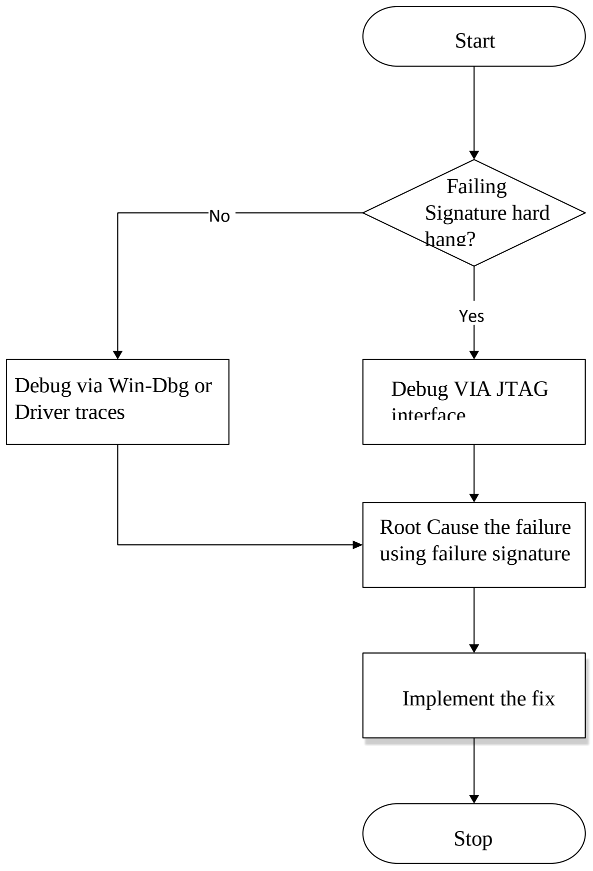

Start
Failing Signature hard hang?
No
Yes
Debug via Win-Dbg or Driver traces
Debug VIA JTAG interface
Root Cause the failure using failure signature
Implement the fix
Stop

Figure. 5 Debug flow for Hibernation issues

For any platform, the flow defined can help and pin point what are the use cases impacted in power management domain and also the effect of same on the corresponding overlapping domains. Issue debug with the flow chart provided helps in promptly resolving and nailing the issue. This would get us also the details on the overall system coverage. Also with the methodology that we are following, it becomes easier for us to check for the hard hangs or soft hang and also follow the proper bucketing procedure as stated earlier.

Let us take a peculiar example of failure and then map it back in out flow chart and then subsequently take it further down to the abstraction level where we see the failure being pin pointed. After that we would also check for the interpretations from the coverage viewpoint what can be removed. As per the description from section IV the major thing that we have failures with hibernate flow is entry and exit from it.

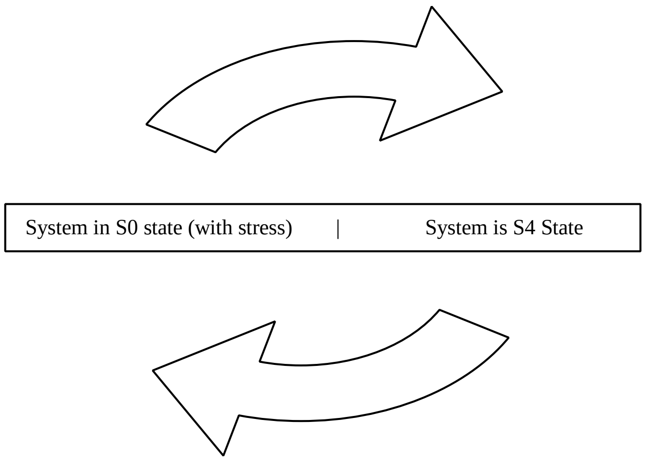


Figure 6. Representation of iterative stress validation scenario

We have a stress testing scenario as shown in Figure. 6 where there are back to back S4 cycles needed for qualification of platform release. During overnight stress run we see the failure, depicting a display off symptom, while system has power and other peripherals are properly in place. Then, we need to start debugging from the point where we need to identify whether the failure was while resuming or entering to the S4 state. After initial level of analysis as per the debugging flow say we zero down to the conditions saying it is soft hang with indication that while entering to S4, system went into corrupt state. After some more analysis we get to know that because of an issue with inbuilt OS drivers we are having a suspected failure.

Inferring from the above information at hand we can tell that we do not have issues with the hardware per say, be it silicon or board specific or any other third party hardware IP. We could also say that there are bunch of probable causes from the software side when we are in process of debugging. Deeper dive in the debug can then in-turn reveal, what is the exact component / entity failing and would help in getting what features are blocked owing to this failure. Once that information is available, we could then get a reduced platform test coverage as the tests involving resuming from S4 in any way would all get blocked. To quantify it we can explicitly say approximately a test plan would see reduction

For better understanding the Figure 7 depicts the pictorial representation of the use case and also gives an idea about inferences at various levels.

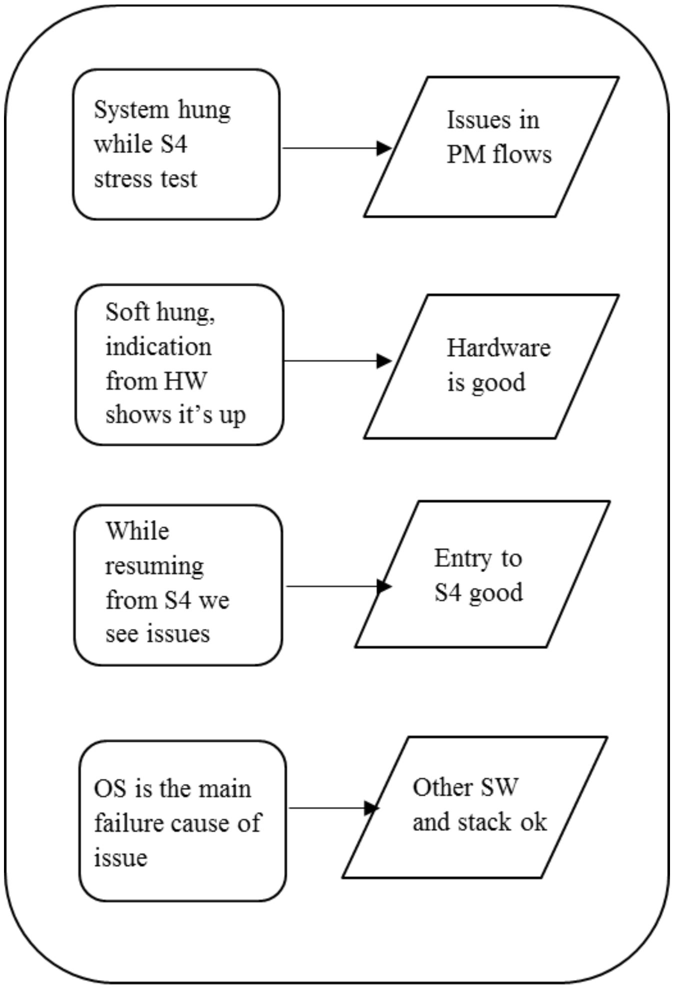

System hung while S4 stress test
Issues in PM flows
Soft hung, indication from HW shows it's up
Hardware is good
While resuming from S4 we see issues
Entry to S4 good
OS is the main failure cause of issue
Other SW and stack ok

Figure 7. Observations and inference matrix generated from S4 fail analysis

From a single issue we could get many inferences and also a fair bit of idea as to what would be main features blocked. In similar fashion we would be having information from all issues coming in and giving the validation teams a clear picture of what is status of platform health, where there are more bugs and more focus is needed and also we could get information on redundant tests that are not yielding bugs. Basically dynamically changing the test plan. So all in all this would emphasize and enhance the test plan quality and also the way system validation is done providing all the necessary aids and opportunities of improvement.

VI. RESULTS & CONCLUSION

This paper discusses the methodology for alignment of system test content and gauging of the details of system coverage, how issues can be debugged efficiently and effectively. The overall essence of paper is to move from the feature centric validation and debug to the use case based approach & intuitive debug of the issues targeting platform agnosticism. The use case approach eventually helps in the easier feature test additions and also the validation at system level. Debugging and error categorization also becomes way easier if we follow this methodology. The proposed methodology can be extended to any of the systems use case wherein we can perform the respective scaling of test plans and other features checks depending on the user scenarios. A running use case example and the debug fan outs for the erroneous conditions are also presented as part of the paper.

As part of deployment of this methodology internally we have used the same approach for the previous two platforms for validation and have seen 20% reduction in validation cycle times overall. If we translate it to direct $$ savings it would be around the 20% budget saving given for the platform validation. This when clubbed with various OS where individual platform validation cycle is performed amounts for a considerable amount of money. Additionally taking this methodology and furthermore AI based algorithms we have developed tools internally which takes the platform defects as inputs and provides us with the requisite test plan generated dynamically, which is a reduced set list depending on the defect trends seen in the earlier runs.

REFERENCES

[1] N. Kumar, "IoT architecture and system design for healthcare systems," *2017 International Conference on Smart Technologies for Smart Nation (SmartTechCon)*, Bengaluru, India, 2017.

[2] A. V. Ramesh, S. M. Reddy and D. K. Fitzsimmons, "Airplane system design for reliability and quality," *2018 IEEE International Reliability Physics Symposium (IRPS)*, Burlingame, CA, USA, 2018.

[3] Advanced Configuration and Power Interface Specification, Version 5. November 2013.

[4] IEEE Standard for Test Access Port and Boundary-Scan Architecture - Redline," in IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001) - Redline, vol., no., pp.1-899, May 13 2013.

[5] J. Ryser, M. Glint A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts

[6] I. Jacobson: Basic Use Case Modeling; Report on Object Analysis and Design, vol. 1, n° 2, pp. 15-19, 1994

Learning to Categorize Bug Reports with LSTM Networks

Kaushikkumar D. Gondaliya
Intelligent Autonomous Systems Lab
Technical University Darmstadt, Germany
email:kaushik.gondaliya1@gmail.com

Jan Peters
Technical University Darmstadt
Max-Planck Institute Tuebingen
Darmstadt and Tuebingen, Germany
email:mail@jan-peters.net

Elmar Rueckert*
Robotics and Cognitive Systems
University of Luebeck, Germany
* Corresponding author
email:rueckert@rob.uni-luebeck.de

***Abstract*—The manual routing of bug reports to specialized expert teams is a time-consuming and expensive process. In this paper, we investigated how this process can be automated by training deep networks and state-of-the-art classifiers from thousands of real bug reports from a software company. Different combinations of the natural language processing methods *lemmatization*, *pos tagger*, *bigram* and *stopword removal* were evaluated in the classification algorithms *Linear Support Vector Machines* (SVMs), *multinomial naive Bayes*, and *Long Short Term Memory* (LSTM) networks. For feature processing we used the *Term Frequency-Inverse Document Frequency* (TF-IDF) method. Best results were obtained with a combination of the *bigram* method and linear SVMs. Similar prediction performance values were observed with LSTM networks that however promise to improve further with larger datasets. The bug triage tool was implemented in a microservice architecture using docker containers which allows for extending individual components and simplifies applications to other text classification problems.**

***Keywords*–classification of text; bug reports; natural language processing; long short term memory networks; support vector machines.**

I. Introduction

The demands of quality software products have rapidly increased and a significant amount of cost is spent on support and maintenance. Although various testing methods are used to ensure a high quality of a software, it is almost never perfect and needs to be maintained continuously. As a result software developers and teams of experts are often confronted with a stream of service requests or bug reports [1, 2]. By a bug report, we refer to code errors or misbehaviour of a software based on an error in a computer program.

These reports have to be resolved by a selected team or expert. Due to the complexity of the reports, most software companies rely on human experts to assign them. This is an expensive task where for example the company who provided us with the data for this research received more than 6.8 million service requests for a single product in 2013. In a study of the *Eclipse project* conducted by Anvik et al. [3], it was found that on average 37 bugs are submitted per day. Moreover, three person-hours per day are required for performing the bug report assignment manually. Therefore, an automatic and efficient bug tracking tool is required to reduce human efforts or to make a bug tracking process less time consuming.

Existing learning approaches applied to such problems are based on training support vector machines or naive Bayes classifiers [4, 5]. While both approaches were shown to produce good results on small datasets [6] or in binary classification tasks [7], their application to large datasets or to online learning tasks is limited. For this problem domain, deep learning algorithms such as *Long Short Term Memory* (LSTM) networks [8, 9] are promising alternatives which have not been used for classifying bug reports so far. We choose LSTMs because they can be trained from small datasets with less than thousand samples in contrast to alternative deep learning approaches like [10]. This benefit was shown in our evaluation.

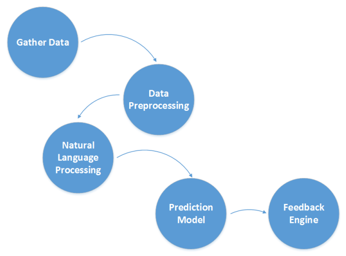


Figure 1. **Overview of the Framework:** Five docker containers are used for automated bug tracking with LSTM networks.

In this paper, we compare state-of-the-art bug assignment approaches to approaches based on LSTM networks. We present thorough results on datasets of real bug reports and validate the models' predictions with feedback collected from experts responsible for resolving the bug reports. Our findings and models can be easily extended and transferred to other text categorization problems.

In the following section, we review related work on automated bug tracking systems. In Section III, we discuss the used methods and In Section IV, we evaluate them on two datasets created from the bug reports provided by a software company. We conclude in Section V.

II. Related Work

Previous research investigated the detection of bug reports [11, 12], bug prioritization [13, 14], bug categarization [4, 5, 6, 7] and severity [13, 15]. In this report, we only focus on categorization and briefly review related approaches.

An important related by Xuan et al. [16] studied automatic bug triage systems based on feature selection and decision trees on the *Mozilla* [17] and the *Eclipse* [18] data sets. A good overview over data preprocessing strategies and feature modeling techniques is given. This work is based on a prior study from [19]. The discussed natural language processing techniques goes beyond the basic methods used here. However, in contrast, we focused on the learning of the classifiers and compared LSTM networks to state of the art naive Bayes and SVMs.

In [4], the authors proposed an automatic routing system to classify incoming bug reports. The goal was to develop a continuously running router with a low misclassification error. The authors gathered around 6000 reports from a large software system which were classified into eight different categories by human experts. They compared several classification approaches like *naive Bayes*, *Support Vector Machines* (SVMs), *classification trees* and *k-nearest neighbor classification*. Their empirical results showed that the probabilistic models, i.e., the *k-nearest neighbor classification* and the SVMs outperformed the others. Furthermore, the accuracy improvement with an increasing amount of training data. However, for natural language processing only the *stopword removal* and *stemming method* were evaluated.

In [5] SVMs were used to train classifiers from two datasets, i.e., the open-source *Eclipse* [18] and open *Firefox* [17] bug report collections. These datasets contained 8.655 and 9.752 bug reports. For the *Firefox* dataset, the developer who submitted the last patch was used for labelling the bug reports. For the *Eclipse* dataset, the developer's name was used for labelling the bug reports, one who marked the bug report as ”resolved”. In a case of duplicates, a name of the developer who resolved the original bug report was taken for Eclipse and Firefox. They achieved an accuracy of 64% for the *Firefox* dataset and a precision of 58% for the *Eclipse* dataset. However, they obtained only a recall value of 3% for the *Firefox* dataset and 10% for the *Eclipse* dataset. For feature selection only the *bag of words method* and the *stopwords removal* approach were evaluated.

In a related approach, [6] evaluated the approaches *naive Bayes*, SVMs, *Radial Basis Function* (RBF) networks and *Random Forest* on a dataset from *Mozilla* [17] containing 1.983 bug reports. They used a *Term Frequency - Inverse Document Frequency* (TF-IDF) weighting scheme for feature extraction. The best result obtained was a classification accuracy of 44.4%, a precision of 30% and a recall value of 28%.

In [7], the authors proposed using the TF-IDF feature extraction method in combination with a *naive Bayes* classifier. They evaluated their approach on a dataset containing 10.000 reports and classified them into security or non-security related reports. On this binary classification problem, the authors achieved an accuracy of 93.9% and a precision of 92.5%. In this work however, we focus on multi-class assignments and the results are not directly comparable.

[6] and [7] showed that the TD-IDF method improves the results for classification tasks. Therefore we build on these results and used the TD-IDF method as well. However, in addition we trained *Long Short Term Memory* (LSTM) networks [8, 9] and tested other natural language processing methods such as *lemmatization*, *pos tagger* and *bigram*.

III. Methods

In this section, we discuss the necessary parts of an automated bug assignment system. These parts are (i) natural language processing techniques that transform and normalize text into a machine learning friendly form, (ii) feature extraction approaches that convert text into vectors of numbers, and (iii) classification algorithms. An overview of how these methods are used for computing predictions of bug report assignments is sketched in Figure 2.

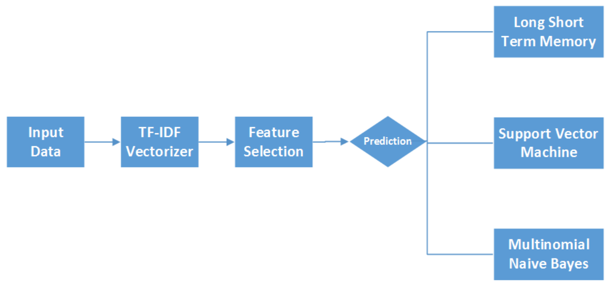


Figure 2. **Prediction model:** Features are generated from the raw text using the TF-IDF vectorizer method. Three classification algorithms, i.e., LSTM, SVM and *naive Bayes* as prediction model.

A. Natural Language Processing Methods

Natural Language Processing (NLP) is used to analyze, understand, and process meaning from human language. NLP is used to perform tasks, such as automatic summarization, translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, or topic segmentation. Some of the most important NLP methods, which are also used in this study are discussed here. For a more detailed review, we refer to [19, 16].

Tokenization.: Text is treated as a string that is chopped into the pieces called tokens. For example, the string ’winter is coming’ is tokenized into the terms ’winter’, ’is’, ’coming’.

Word Boundary.: It removes the extra white spaces or punctuation from the text. However, this removal entirely depends on the domain and for that reason in most cases regular expressions are used. For example, punctuation in terms like in ’M.Sc.’ might contain information depending on the context.

Lemmatization or Stemming.: The document could consist the same words in many forms such as infection, derivation, etc. Due to grammatical reasons *lemmatization* and *stemming* are used to convert different word variants into similar canonical forms. For example, the different forms of the words ’work’, ’works’, ’worked’ and ’working’ share a same stem ’work’.

Stopword Removing.: Commonly used words like ’a’, ’the, ’of’ etc. normally do not contain any meaningful information. So it is beneficial to remove these words from the text.

Part-of-Speech Tagging.: The *POS tagger method* reads the texts from the document and applies labels like ’noun’, ’verb’, etc. to them. For example, the string ’Heat water in a large vessel’ will create the tags ’(heat,VB), (water,NN), (in,IN), (a,DT), (large,JJ), (vessel,NN)’.

N-gram.: Sometimes, a group of words is more beneficial than just a single word. Here, N is the number of words in a group. When N=1 the approach is called unigram, with N=2 it is called bigram, and with N=3 it is called trigram. An example for a bigram of the string ’winter is coming’ is ’Winter is’, ’is coming’.

We evaluated two basic pre-processing strategies commonly used in the literature [19, 16]. These two approaches

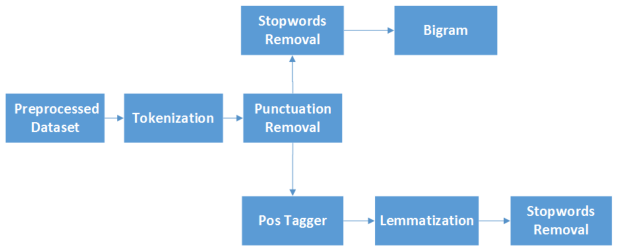


Figure 3. **Features:** We evaluated two sequences of text pre-processing techniques commonly used in text categorization.

are shown in Figure 3. The first strategy terminates the pre-processing with the *bigram* method. In the second sequence of applied NLP techniques, we end with the *stopword removal* method. For the remaining part of this paper, we will denote words in the pre-processed text by the variable w.

B. Feature Extraction

In NLP, the most widely used extraction methods are the *Term Frequency - Inverse Document Frequency* (TF-IDF) [20], the *word 2 vector* method [21] and the *count vectorizer* approach [22]. We used the TF-IDF approach which was shown to outperform the other techniques in related applications [6, 7].

The term *TF* in the TF-IDF approach denotes the frequency count of keywords t which is weighted by its importance denoted by the term *IDF*. Thus, a feature in the TF-IDF approach results from the product of the TF count with the IDF weight, i.e., TF-IDF(t) = TF(t,d) · IDF(t). The term-frequency TF is defined as

$$\mathrm{TF}(t,d) = \sum_{w\in d} \mathrm{f}(w,t),$$

where t denotes a keyword and w a word in text d. The function f returns 1 for a match ($t = x$) and zero otherwise. Now, given a dataset of K documents $D = \{d_1, d_2, \ldots, d_K\}$, the inverse document frequency (IDF) is defined as

$$\mathrm{IDF}(t) = \log \frac{K}{1 + |\{d : t \in d\}|}.$$

Here the term $|\{d : t \in d\}|$ denotes the cardinality of documents, i.e., when the condition TF$(t,d) \neq 0$ is satisfied. Note that 1 is added to the denominator to avoid a divide by zero situation.

Finally, for each bug report in dataset D a feature vector for classification $\mathbf{x} = [x_1, x_2, ..., x_N]^T$ is computed by evaluating x_n = TF-IDF(t) for each of the selected keywords denoted by t.

C. Classification Algorithms for text

In this section, we discuss the commonly used *naive Bayes* classifier, *Linear Support Vector Machines* and *Long Short Term Memory* (LSTM) networks for text classification.

1) Naive Bayes Classifier: The *naive Bayesian classifier* is a generative linear model based on the *Bayes theorem* [23, 24]. Important to note is that it relies on the assumption that all the features are independent.

In the context of text classification, the probability that text T belongs to a class C can be expressed as $P(C|T)$, where

$$P(C|T) = \frac{P(C)P(T|C)}{P(T)}.$$

The distribution $P(C)$ denotes the prior probability of class C, $P(T)$ is the prior probability of text T and $P(C|T)$ is posterior probability that we need to compute. Given vectorized feature representations of text of the form $\boldsymbol{x} = [x_1, x_2, ..., x_N]^T$ the above equation can be interpreted as:

$$\begin{aligned} P(C|T) &= P(C|\boldsymbol{x}), \\ &= P(C|x_1, x_2, ..., x_N), \\ &= \frac{P(C)P(x_1, x_2, ..., x_N|C)}{P(x_1, x_2, ..., x_N)}. \end{aligned}$$

Assuming independent features for a given class the posterior distribution above factorizes to $P(C|T) \propto P(C)\prod_{k=1}^{N} P(x_k|C)$.

For a new document in a test dataset, we can compute class labels using the maximum a posteriori decision rule [25],

$$C_{map} = \underset{x\in C}{\operatorname{argmax}}\, P(C) \prod_{k=1}^{N} P(x_k|C),$$

where $P(C)$ is the prior probability of class which can be estimated as follows:

$$P(C) = \frac{\#\ of\ instances\ in\ this\ class}{\#\ of\ instances\ in\ all\ classes}.$$

The presented *naive Bayes* approach is used for multi-class classification in our experiments.

2) Linear Support Vector Machine: SVMs are powerful and popular supervised learning approaches [26]. They can be used for both classification and regression problems though they are mostly used for classification problems. SVMs are applicable for both linear and nonlinear classification problems using kernels. However, according to related work, see for example the discussion in [20], linear support vector machines are sufficiently powerful for text classification problems.

For samples $\boldsymbol{x_k}$, with class labels $c_k \in \{-1, 1\}$ for $k = 1, ..., N$, we compute a hyperplane which satisfies $\boldsymbol{v}\,\boldsymbol{x_k} + b = 0$. Here, $\boldsymbol{v}$ is a vector orthogonal to the hyperplane and b is a perpendicular distance of the hyperplane to the origin. The canonical hyperplane is observed when the condition $c_k(\boldsymbol{v}\,\boldsymbol{x_k} + b) \geq 1\,\forall k$ is met.

For classification problems with more than two categories the most frequent approaches are the *One-vs.-Rest* method and the *One-vs.-One* method. In this paper, the *One-vs.-Rest* classifier is used for multi-class classification.

D. Long Short Term Memory Networks

LSTMs are recurrent neural networks which were proposed by Hochreiter et al. [8] to overcome the vanishing gradient problem. The ability to capture temporal correlations over long time horizons was exploited in many speech processing and deep learning applications [9]. For text categorization however, we are not aware of any study using LSTM networks for computing multi-class label predictions.

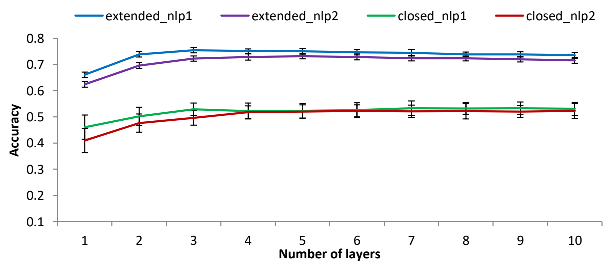


Figure 4. We tested different numbers of hidden layers for two pre-processing strategies. Results are shown for a small dataset of 1215 bug reports and on an extended one with 7346 reports.

For text classification, the LSTM network is trained on a sequence of feature vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N$ that update the internal hidden state denoted by $\boldsymbol{h}_k$. The corresponding output is trained to fit $\boldsymbol{c}_k$. In this paper, we used sigmoid activation functions for the input layer. We optimized for the optimal number of hidden layers, i.e., 3 and 6 layers for the two evaluated datasets as shown in the right panel in Figure 4.

For the experiments in this paper, we used an open-source framework implementation (*Keras*) with a training batch size of 32. The neural network model had two hidden layers with 64 neurons in the first layer and seven units with soft-max activation functions in the output layer. A dropout regularization of 25% was used to reduce the models complexity and to prevent over-fitting. For training the *categorical cross entropy loss* function was used.

IV. Experimental Results

In this section, we present results of the classification models on two datasets created from the bug reports provided by a software company. We prepared a small dataset with 1215 reports and a large dataset with 7346 samples, which is illustrated in Figure 5. The bug reports in the small dataset were already processed by the expert teams and had therefore labels or class assignments. For the large dataset we had to generate labels. The denote this manual labelling process by the *keyword algorithm* for the remainder of this paper.

Manual data labelling: To generate a larger dataset, a *keyword algorithm* was used to manually generate labels. As first step, experts assigned keywords to just submitted and therefore unprocessed bug reports. The algorithm looks for matches to fixed sets of keywords dedicated to each of the specialized support teams. Labels were thereafter created based on the maximum number of occurrences of experts' keywords. In case of equal counts, a random team assignment was generated.

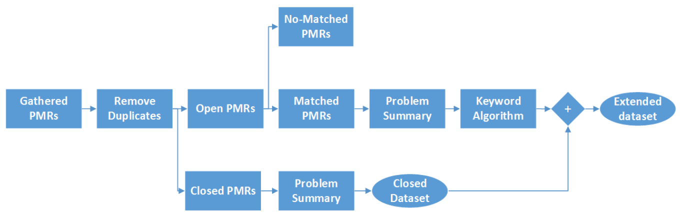


Figure 5. **Data preprocessing:** The execution of a process on gathered data results in ’closed’ and ’extended’ dataset.

We first present results evaluating the effect of the number of features. Subsequently we discuss the effect of different combinations of pre-processing and classification techniques in the two datasets. We conclude by verifying the assumption of the creation of the additional labels for the large dataset. For that human experts rated the predictions of our models. All presented statistics were obtained through running 100 experiments with 79% of the samples randomly selected as training set and 21% as test set.

A. The effect of the number of used features

We used the *Term Frequency-Inverse Document Frequency* (TF-IDF) method which is discussed in Section III-B for feature selection. In Figure 6, we show the classification performance values for the three classification methods SVM, LSTM and naive Bayes (NB) for an increasing number of used features. The *stopwords removal* and the *bigram* methods were used for the pre-processing of the bug reports.

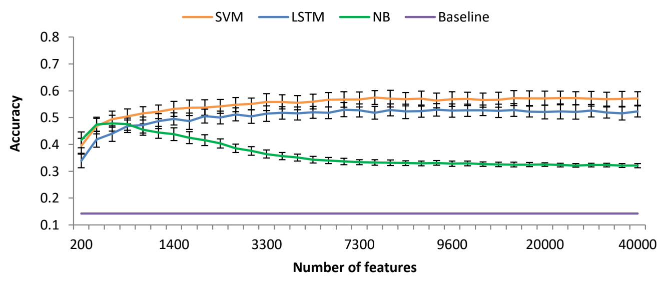


Figure 6. Evaluation of the effect of the used number of features in three classification methods. The baseline accuracy for a 7-class classification problem is $1/7$. The classifiers were trained on the small dataset.

According to Figure 6, we observe that a linear SVM outperforms the *multinomial naive Bayes* and the LSTM network in general. The best accuracy values for *multinomial naive Bayes*, linear SVM and the LSTM network obtained were 0.479, 0.574 and 0.529 with 600, 7900 and 6700 features respectively.

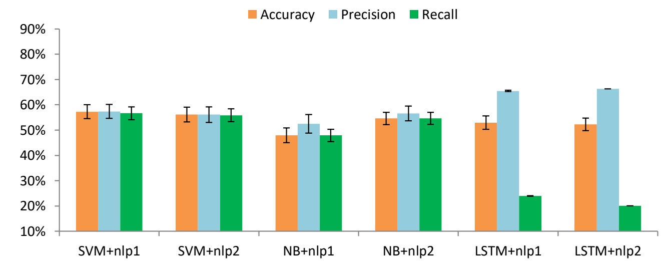
B. Comparison of the Prediction Models

On the small dataset all three classification methods correctly classified $47.9 \rightarrow 57.2\%$ of the test samples. Note that these results are significantly better than random assignments (14.3% for seven classes). For the large dataset the performances ranged from $68.6 \rightarrow 77.6\%$. For this experiments the optimal number of features for each classifier was used as evaluated in the previous subsection.

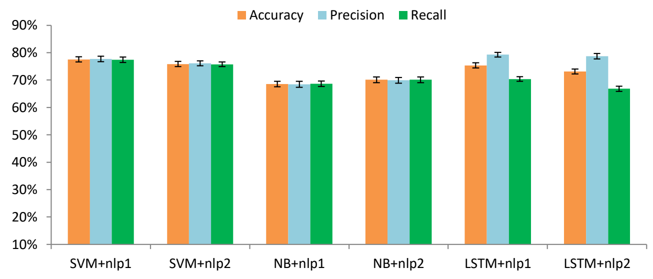
We also evaluated the effect of two commonly used pre-processing approaches that were discussed in Subsection III-B. With the term ’nlp1’ we denote the application of the *stopwords removal* and the *bigram* methods and with the term ’nlp2’ we denote the the combination of the *pos tagger*, *lemmatization* and *stopword removal* methods. As shown in Figure 7, no significant difference in the classification performance could be found. However, the orchestration of methods denoted by ’nlp2’ is favoured because of computational reasons.

C. Verification of the automatically labeled bug reports

The ’large dataset’ relied on the assumption that correct labels could be automatically created using a keyword algorithm. The keywords were provided by experienced support engineers who were also asked to validate the predictions of our three trained classifiers. A feedback engine was implemented and



(a) Results for the small dataset.



(b) Results for the large dataset.

Figure 7. Shown are the classification performances for the combination of *linear SVM*, *multinomial naive Bayes*, and LSTM networks with different pre-processing approaches. The terms ’nlp1’ and ’nlp2’ denote two commonly used preprocessing strategies, see the text for details.

the experts rated our models’ predictions of additional 132 bug reports which were not used for training.

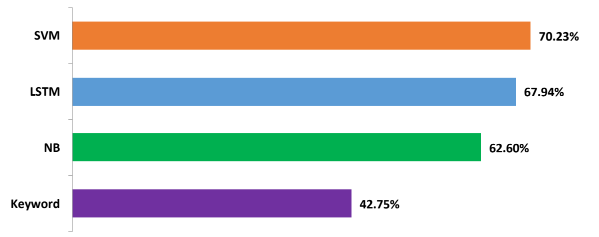
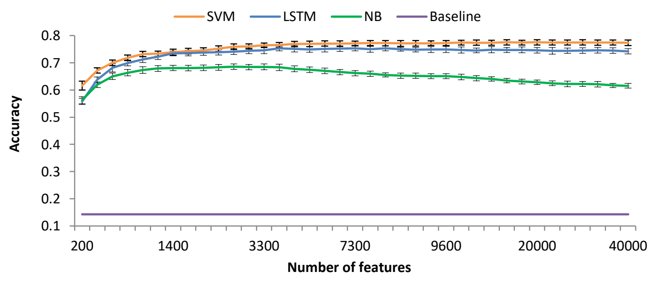


Figure 8. The experts’ feedback values for the linear SVM, *multinomial naive Bayes*, and LSTM networks and ’keyword algorithm’.

The results are shown in Figure 8. For SVMs 70.23% of these new reports were correctly assigned to one of the seven teams. For LSTMs 67.94% of the assignments were correct. The naive Bayes approach generated 62.6% of correct labels. Note that the predictions of the keyword algorithm could only classify 42.75% of the reports correctly. These predictions and results were perceived as very helpful and will safe substantial resources in the future for performing bug report assignments.



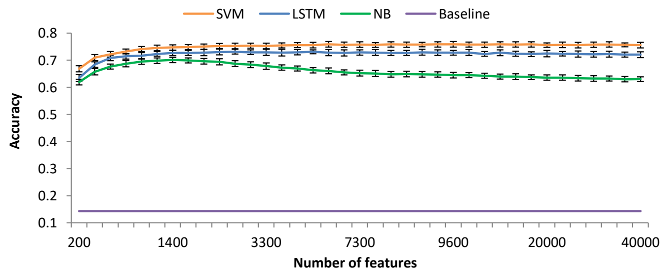


Figure 9. The prediction accuracy improves with the number of used features. Left: ’nlp1’ and right ’nlp2’ preprocessing approach.

D. Specific Feature Selection.

We also evaluated accuracy values for different number of features by applying *Term Frequency-Inverse Document Frequency* (TF-IDF) method. For the ’large dataset’ the results are shown in Figure 9.

V. CONCLUSION

Every year software companies receive millions of service requests that have to be resolved by specialized expert teams. The assignment of the requests is traditionally done by human experts. First attempts in automatic routing of service requests are based on training *Support Vector Machines* (SVMs) and *naive Bayes* multi-class classifier [4, 3, 7]. However, these approaches are computationally and memory demanding for large datasets and thus of limited practical use for large software companies.

In this work, we investigated the performance of *Long Short Term Memory* (LSTM) networks which can be trained from millions of samples. We evaluated different combinations of natural language processing methods such as *lemmatization*, *pos tagger*, *N-gram* and *stopword removal* and tested different numbers of features ranging from 200 to 40.000. We found that for small datasets with 1215 reports, SVMs achieved the best classification performance. Out of 256 bug reports of a test set, $57.2 \pm 0.028\%$ were correctly assigned to one of the seven expert teams. In contrast with LSTMs only $52.9 \pm 0.026\%$ were correctly assigned. Here, the $\pm$ symbol denotes the standard deviation computed from 100 runs.

Interestingly, with an increasing number of training samples LSTM networks achieved similar classification results. This was shown in training from a larger dataset with 7346 samples. Here SVMs correctly classified $77.6 \pm 0.009\%$ and LSTM $75.3 \pm 0.009\%$. Given that LSTM networks were used with millions of samples in deep learning approaches we expect them to outperform SVMs with larger datasets. This assumption will be tested in near future as the framework is constantly used to collect new service requests, i.e., during the time writing this report 2000 additional request were obtained. Moreover, we plan to compare our pre-processing methods, feature extraction strategies and deep network classifier to different text categorization problems.

Acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreements No 713010 (GOAL-Robots) and No 640554 (SKILLS4ROBOTS). The authors also thank the software company who provided the data.

References

[1] G. Sharma, S. Sharma, and S. Gujral, "A novel way of assessing software bug severity using dictionary of critical terms," *Procedia Computer Science*, vol. 70, pp. 632 – 639, 2015.

[2] B. Beizer, *Software Testing Techniques*. Dreamtech, 2003. [Online]. Available: https://books.google.co.in/books?id=Ixf97h356zcC

[3] J. Anvik, "Automating bug report assignment," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 937–940. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134457

[4] G. A. Di Lucca, M. Di Penta, and S. Gradara, "An approach to classify software maintenance requests," in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 93–102.

[5] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[6] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine," in *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ser. ICSEA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 216–221. [Online]. Available: http://dx.doi.org/10.1109/ICSEA.2009.92

[7] D. Behl, S. Handa, and A. Arora, "A bug mining tool to identify and analyze security bugs using naive bayes and tf-idf," in *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, Feb 2014, pp. 294–299.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[9] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[10] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning*, 2014, pp. 1188–1196.

[11] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368151

[12] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, ser. APSEC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 366–374. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2010.49

[13] K. Chaturvedi and V. Singh, "Determining bug severity using machine learning techniques," in *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*. IEEE, 2012, pp. 1–6.

[14] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 495–504. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806871

[15] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 1–10.

[16] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, "Towards effective bug triage with software data reduction techniques," *IEEE transactions on knowledge and data engineering*, 2014.

[17] I. mozilla.org contributors, "Bugzilla dataset," https://bugzilla.mozilla.org/, 19982017, retrieved: Aug,, 2018.

[18] T. E. Foundation, "Eclipse dataset," http://www.eclipse.org, retrieved: Aug,, 2018.

[19] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 25–35.

[20] T. Joachims, "Text categorization with suport vector machines: Learning with many relevant features," in *Proceedings of the 10th European Conference on Machine Learning*, ser. ECML '98. London, UK, UK: Springer-Verlag, 1998, pp. 137–142. [Online]. Available: http://dl.acm.org/citation.cfm?id=645326.649721

[21] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation." in *EMNLP*, vol. 14, 2014, pp. 1532–43.

[22] A. Tripathy, A. Agrawal, and S. K. Rath, "Classification of sentimental reviews using machine learning techniques," *Procedia Computer Science*, vol. 57, pp. 821–829, 2015.

[23] R. O. Duda and P. E. Hart, *Pattern classification and scene analysis*. New York: John Wiley, 1973.

[24] P. Langley, W. Iba, and, and K. Thompson, "An analysis of bayesian classifiers," in *Proceedings of the Tenth National Conference on Artificial Intelligence*, ser. AAAI'92. AAAI Press, 1992, pp. 223–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=1867135.1867170

[25] C. D. Manning, P. Raghavan, and H. Schutze, "Text classification and naive bayes," *Introduction to information retrieval*, vol. 1, p. 6, 2008.

[26] V. N. Vapnik, *The Nature of Statistical Learning Theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.

Improving Testability of Software Systems that Include a Learning Feature

Lydie du Bousquet
Univ. Grenoble Alpes,
CNRS, Grenoble INP, LIG,
38000 Grenoble, France
email: Lydie.du-Bousquet@imag.fr

Masahide Nakamura
Graduate School of System Informatics,
Kobe University, Japan,
RIKEN AIP.japan
email: masa-n@cs.kobe-u.ac.jp

***Abstract*—In this article, we describe the work we did to validate a case study application that includes a learning feature. Our objective was to express properties that can be used for testing or monitoring the quality of the application, taking to account the learning dimension. To express relevant properties, we had to modify the architecture of the application to add supplementary outputs. They indicate whether the learning phase is achieved or not, and if the system notices some significant changes in the environmental conditions. Here, we report on the lessons learnt about this process.**

***Index Terms*—Learning System; Validation; Testability; Design for Testing**

I. Introduction

Nowadays, Artificial Intelligence (AI) is becoming more and more important in software systems. Learning ability for new apps or systems is a promise of a personalised experience for every user based on individual preferences. Thus, the user's satisfaction is expected to be improved.

For example, authentication facilities are now offered for smart-phones that learn and recognise the user's face [1]. Virtual personal assistants are able to learn the user voice to satisfy the voice commands. AI systems send recommendations for shopping [2]. Self-driving cars are travelling more and more kilometres.

However, safety or security problems are regularly reported for those applications. For example, researchers were able to send secret audio instructions undetectable to the human ear to Apples Siri, Amazons Alexa and Googles Assistant [3]. Tay, a chatbot created for 18- to 24- year-olds in the U.S. for entertainment purposes, caused subsequent controversy when the bot began to post inflammatory and offensive tweets through its Twitter account, forcing Microsoft to shut down the service only 16 hours after its launch [4] [5]. A woman was killed by a self-driving car in Arizona in March 2018 [6].

Those reported problems make AI more suspicious for users and increase the need for validation, verification, and even certification. However, learning features make software system much more difficult to validate [7]. Indeed, while learning the user's habits or the environmental characteristics, the system behaviour is evolving in an unpredictable way [5]. Specification becomes somehow tricky to express, which impacts the validation procedures.

In this article, we report the lessons learnt from the validation of a case study application that includes a learning feature. To be able to express properties that can be used for testing or monitoring, we had to modify the design of the systems. In Section II, we first report on some related works dedicated to the validation of machine learning algorithms and systems that include such algorithms. We then detail our case study (Section III) and the expression of properties (Section IV). We discuss the lessons learnt in Section V.

II. Machine learning and validation

In this section, we successively consider the validation of Machine Learning algorithms, the validation at the system level and the testability point of view.

A. Validation of the algorithms

There is a large variety of Machine Learning (ML) algorithms dedicated to different applications. For example, classification ones are used for classifying discrete inputs into predefined categories. Clustering algorithms are used to group similar inputs (into clusters). Regression analysis is used for prediction and forecasting.

A ML algorithm is expected to predict well after training. This property is called generalisation. The first step of the validation is thus to check the quality of the prediction, also called *performance* of the ML algorithm. This can be compared to the functional validation step in software engineering, which aims to ensure that a program is doing well what it is supposed to do.

Performance measures are specific to the algorithm class that is considered. For pattern recognition and binary classification, it is evaluated through *precision*, *recall* and *F-measure* (the weighted harmonic mean of precision and recall) [8]. They are based on the count of true/false positive/negative verdicts. True (resp. False) verdict are used to differentiate when the answer of the system recognises (or not) the input. Positive (resp. negative) verdict states whether the answer is correct or not. For algorithms such as classification and regression, *generalisation error* can be evaluated to measure how accurately the algorithms are able to predict outcome values for previously unseen data [9].

Other properties can also be expected for ML algorithms. *Stability* evaluates how much a ML algorithm is perturbed by small changes to its inputs: the predictions of a stable algorithm will not be very affected if the inputs change just a little bit [10]. For a large class of learning algorithms, notably empirical risk minimisation algorithms, certain types of stability ensure good generalisation. *Training speed*, *efficiency*, *compactness* are also dimensions that can be evaluated, especially to compare different algorithms.

Several *methods* are proposed to evaluate the quality of the algorithms. In holdout evaluation, data available for training are split into two sets, one for the training itself, the second one for the validation [11]. This can be done randomly or may involve more complex sampling methods.

Cross-validation denotes a set of more sophisticated validation methods. Basically, they consist in dividing the dataset into equally sized groups of instances (called folds). The learning algorithm is then applied several times, each time using the union of all subsets but one, which is used as a test set [12]. The cross-validation methods are different from one another by the way to build the folds and to use them. To produce the learning set, some test methods can be applied [13] [14].

Learning and validation with cross-validation approaches can give very good results. However, they are mainly dedicated to situations *"in the lab"*. For *"in the wild"* cases, i.e., for real-world situations where applications learn from the final user after installation, simpler approaches might have to be applied due to lack of time or data. Poorer results are observed [15]. For this reason, it is necessary to consider also the quality of the final system considered as a whole.

B. Validation of the final system

Validation of a system that includes some learning feature is difficult. As said previously, the final usage of the application is not easily predictable since it varies with respect to the environment. Sometime, even for one specific user, her needs may evolve with the time. This makes *specification* impossible to express with precision [7] [16] . For the same reasons, it is usually not possible to assess the *environment* characteristics of the final system precisely.

In [17], authors use *simulations* to evaluate the quality of an online adaptive system for electric wheelchairs that learn to avoid obstacles. For this specific application, some safety properties are easy to express, e.g., the wheelchair should not enter in collision with any obstacles. However, the quality of the learning is more difficult to assess with a property. So, the authors evaluated the path smoothness of the wheelchair manually in different environments, based on a *comparison* of two algorithms.

In [18], authors focus on the validation of learning features embedded in applications dedicated to intelligent inhabited environments. They performed experiments in which an intelligent application learned and adapted itself to the user behaviour, while she stayed in a real equipped flat for five days. Here again, the quality of the learning feature is assessed by a comparison of the results of different algorithms.

Model-checking approach has been used in [19]. Authors focus on the safety and robustness validation of vision systems that can be used for self-driving cars. One of the considered safety properties is that "self-driving cars steering angle should not change significantly for the same road under different lighting conditions". To evaluate this invariant, authors propose a framework that transforms a given image with different transformation functions (e.g., rotation). The objective is to assess the quality of the final application (after learning) by analysing how often the invariant is violated.

In [20], authors advocate the usage of quantitative verification at run-time to identify and even predict requirement violations, in addition to off-line verification for the self-adaptive feature, but it is not clear how to adapt them for specifying properties on learning.

In [21], authors advocate the usage of metamorphic testing to ease the problem of the oracle expression ML applications. Metamorphic testing aims at creating new test cases from the existing ones thanks to a transformation. Well chosen, the transformation approximates the expected outputs of the new tests based on the expected outputs of the old ones.

Beyond the expression of the expected properties (and/or the metamorphic relations), system testing remains difficult to apply, especially because it requires to control the input of the system [22]. For instance, testing an intelligent home application that regulates the home temperature may require to be able to modify the home temperature by other means, to check that the system under test reacts correctly. Overheating a room in winter to check that the air-conditioner can cool it correctly is often unacceptable.

For this reason, the final system is often tested in a simulated environment before deployment [23]. For a validation after deployment, monitoring is often preferred [24] [25]. It consists in observing the outputs of the system without controlling the inputs.

C. A software engineering viewpoint of testability

Testability denotes the ability of a system to be tested [26]. Initially, testability was defined for hardware components. For software systems, several definitions have been proposed. In [27], testability is defined as the effort needed for testing. For Binder, testability is the relative ease and expense of revealing software faults [28]. Other definitions allow a quantitative evaluation of the testing effort [29] or represents the probability to observe an error at the next execution if there is a fault in the program [30].

Being able to characterise and to produce testable systems has become a preoccupation more and more important for software companies. It often means to increase the ability to observe the internal behaviours of the system under test (observability) and/or to increase the ability to control the system under test (controllability).

The case study that follows was a source of reflection for the expression of properties that can be used for testing oracle or

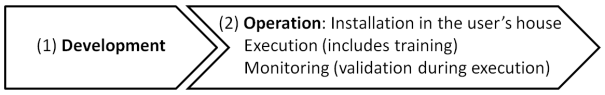

(1) Development
(2) Operation: Installation in the user's house
Execution (includes training)
Monitoring (validation during execution)

FIG. 1. ABSTRACT SOFTWARE LIFE-CYCLE OF THE INTELLIGENT AIR-CONDITIONNER

monitoring. To express them, we had to modify the application design. In the following, we first describe the case study. We then show how the expression properties impacted the design.

III. CASE STUDY

In the following, we present the application under test and the validation harness.

A. The application under test

Our case study is an "intelligent air-conditioner" (iAC). It controls a traditional air-conditioner and offers a planning functionality.

The traditional air-conditioner can be switched on and off. It has three modes: `Cooling`, `Heating` and `Idle`. The AC enters the `Cooling` mode if the observed temperature is beyond the required temperature of more than one degree. It enters in the `Heating` mode if the observed temperature is below the required temperature of more than one degree. It enters in the `Idle` mode when the observed temperature is equal to the required one $\pm$ 0.5. The required temperature can only be set between 17 and 27^{o}C.

The planning functionality allows the user to set a timer value above which the room temperature is expected to be equals to a specified one. To make this possible, the system includes an intelligent feature that learns how long it takes to warm or to cool the room (room thermal inertia).

The life-cycle of the iAC software is depicted at an abstract level in Figure 1. It is composed of two main phases, which denote the time before and after the deployment of the application, and called respectively *development* and *operation* phases. The learning is carried out once the iAC is installed in the user house, i.e. during the operation phase.

This case study is an example of a system "in the wild": it is not possible to achieve training on an pre-existing data set nor is it possible to apply cross validation methods. It is also difficult to achieve testing at system level (i.e., on a real installation) because the environment of the system is hardly controllable for the reasons given Section II-B.

Our objective is thus to achieve monitoring of the system execution in order to check that the system behaviours are adequate. By monitoring, we mean that an external program will periodically check that the inputs and outputs of the iAC satisfy the expected behaviours. We especially want to detect if the learning phase carried out during operation leads to inacceptable behaviours (to prevent situations such as Tay chatbox [5]).

The difficulty relies in expressing what are the adequate behaviours, i.e., to express the right specification/properties. Intuitively, the user expects that the the planing functionality works correctly (the required temperature should be reached on schedule). This means that the learning feature should learn correctly the thermal inertia. Of course, at the beginning, the system can fails, but the errors should be less and less with the time. The user could expect that the learning does not last too long. Another important requirement of the planning functionality is that it is supposed to spend as less energy as possible to achieve the chosen temperature: i.e., it should switch-on the AC just on time w.r.t. the room thermal inertia.

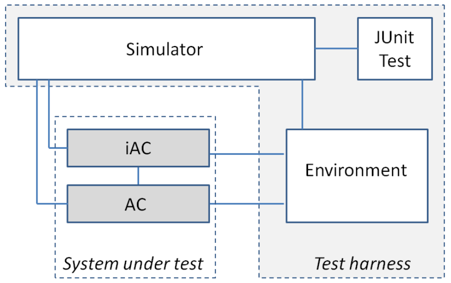


FIG. 2. SIMPLIFIED TEST HARNESS

B. The validation harness

As said previously, our case study aims at providing support to express some expected properties of a system with learning abilities. We use a simulation approach such as in [17] to valid the properties. The validation harness thus consists of a simulator and an environment component, which are controlled by a JUnit test (see Figure 2).

The system under test is developed in Java. The learning feature was implemented as an ad-hoc algorithm. It has no importance by itself, since the system is considered as a black-box for the validation point of view [22].

The environment component aims at simulating the room temperature evolution, which is read by the temperature sensors of AC and iAC. It is possible to modify the room inertia rules during the simulation to fake some environment changes (e.g., outside temperature, opened windows).

The simulator is responsible for the validation progress. It includes:

- *initialisation* methods, to create, initialise and connect the iAC, AC and environment components,
- *property* methods (also called oracle methods), in which expected properties of the system under test are expressed
- a "step" method that deals with the evolution of the system for a given time lapse. During one "step", the simulator checks the state of the AC and asks the environment to update according to equation (1).

$$temp = \begin{cases} temp + \delta_1 \texttt{ if AC is On and Heating} \\ temp - \delta_2 \texttt{ if AC is On and Cooling} \\ temp - \delta_3 \texttt{ if AC is Idle or Off} \end{cases} \tag{1}$$

The simulator is solicited with a JUnit test file, in which it is possible to specify a modification of δ_1, δ_2 and δ_3 during the simulation. In this test file, it is also possible to evaluate the value of the oracle properties as JUnit assertions. Thus, if one of these properties is violated during the simulation, JUnit raises a `Fail`.

IV. EXPRESSING PROPERTIES

Our intelligent AC has to achieve the basic properties expected for an AC. Informally, this means that when the AC is `On`, "it should heat (resp. cool) the room when it is in its `Heating` mode (resp. `Cooling` mode)", and "it is supposed to be in the `Heating` (resp. `Cooling`) mode when the room temperature is substantially below (resp. above) the expected one". Besides, the iAC has to manage the timer feature correctly: the expected temperature should be reached on time. Moreover, it is supposed to spend as less energy as possible to achieve the chosen temperature. In the following, we focus on the validation of the intelligent feature.

Let us first consider P, a property stating that the room temperature should be equal to the expected one when the timer is elapsed. Considering the imprecision of the measure, such a property can be expressed as:

$$(P): \ \texttt{iAC.timerElapsed} \Rightarrow \\ \mid \texttt{env.temp} - \texttt{iAC.requiredTemp} \mid < 0.5$$

where `iAC.timerElapsed` is a Boolean variable that is `true` exactly when the time is elapsed and `false` otherwise, `env.temp` is the observed environment temperature, and `iAC.requiredTemp` is the user required temperature. The imprecision tolerance was arbitrary fixed to 0.5 here without a loss of generality.

To check if the system spends as less energy as possible, it is possible to monitor how often the AC switches from the `Idle` state to an *active* one (`Heating` or `Cooling`), from the moment where the timer is set until it is elapsed. To compute this, we added the `sim.iSwitch` attribute in the simulator component. It is computed like an observer property. Ideally, `sim.iSwitch` value should be 0 when the time is elapsed, but some flexibility could also be acceptable. Indeed having too restrictive properties could provoke unnecessary fail verdicts. For this reason, we chose the following property.

$$(Q): \ \texttt{iAC.timerElapsed} \ \Rightarrow \texttt{sim.iSwitch} \leq 1$$

Moreover, it would be inefficient from an energetic point of view to cool the room just after heating it (or conversely). To capture those situations, in an identical way than previously, it is possible to monitor the number of switches from `Heating` to `Cooling` states (and conversely), and to express a property on the maximum number of changes while a timer is active.

One problem with the previous properties is that they can raise false negative verdicts, i.e., they can be violated even if the system is correct. Three situations have been identified during our tests: (1) the user tries to fix an unfeasible timer, (2) the environmental characteristics have changed during the execution (e.g., a door or window which had been left open), or (3) the training of the system is not achieved.

To fix the situation (1), we modify the iAC program, to make timer activation possible only when the system evaluates that it has enough time to reach the required temperature within the delay. If it has not, a notification is sent to the user, and the AC is switched-on immediately. It has no impact on the former properties because the timer is not activated in this situation.

To fix situations (2) and (3), we need to know whether the training is achieved and if the environmental conditions have changed during the execution. To get that information, we modify the system design to have two new Boolean outputs. The first one, `iAC.stillLearning`, is `true` as long as the system considers that its training is not achieved. The second one, `iAC.envModification`, is `true` if the system noticed significant changes in the environmental conditions while it was trying to achieve the timer requirement, and `false` otherwise.

Thanks to these two outputs, it is possible to rewrite P and Q properties so that a failure occurs only if they are violated after the training end and if the environmental characteristics are the same as those which were learnt.

$$(P'): \ \texttt{not} \ P \ \Rightarrow \ (\ \texttt{iAC.stillLearning} \ \vee \\ \texttt{iAC.envModification} \ )$$
$$(Q'): \ \texttt{not} \ Q \ \Rightarrow \ (\ \texttt{iAC.stillLearning} \ \vee \\ \texttt{iAC.envModification} \ )$$

It is worth noting that `iAC.stillLearning` is an important feedback about the system's learning ability. It allows expressing several properties about the quality of the learning feature. For instance, if the system stays in the state `iAC.stillLearning` after using the timer feature "a lot of" times, it may denote a difficulty. Of course, the acceptable learning time (i.e., number of activations) has to be defined. Let `iAC.nbActivation` be the number of activations of the timer since the iAC installation. The following L property is a way to express a too long training process (5 being chosen arbitrarily):

$$(L): (\ \texttt{iAC.nbActivation} > 5 \ ) \wedge \texttt{iAC.timerElapsed} \\ \Rightarrow \texttt{not iAC.stillLearning}$$

Similarly, `iAC.envModification` can be used to evaluate the learning feature quality. If it is `true` "too often", it may denote that the system is not able to predict the environment behaviours correctly. In the case of our case-study, the room thermal inertia is susceptible to strongly depends on the door's state (i.e., if it is open or close). If the system is not aware of the door status, it may be not able to learn the thermal inertia properly. The analysis of the `iAC.envModification` variations is a possible way to detect such a situation.

It can be noticed that a system that refuses any timer will be correct with respect to the previous properties (because `iAC.timerActivated` will never be set to `true`). To detect such behaviours, we modified the simulator. Each time a timer is set, the simulator captures the notification of the system (acceptance or rejection of the timer). If the system accepts the timer, nothing is done: the previous properties will catch abnormal situations. If the system refuses the timer, the simulator starts a specific timer counter.

If the expected temperature is reached at the end of the timer in normal conditions (`not` (`iAC.stillLearning` $\vee$ `iAC.envModification`)), the system may have been too pessimistic, and a counter value is incremented. It is then possible to express a property that is false if the pessimistic refusal rate is higher than a given threshold.

V. Lessons learnt

With this work, our objective was to assess the quality of the learning feature when embedded in the final system. In this context, the issue is to guarantee that the intelligent part of the system does what exactly it is expected to do: i.e., to provide the right indications to take decisions and/or that it carries out the appropriate actions.

Being able to validate the learning feature is tricky because it is not possible to anticipate the usage and the environmental conditions of the system under consideration. Moreover, classical validation methods such as cross-validation or output comparison of several algorithms may not be possible to achieve. Monitoring the system behaviour is possible as long as expected properties can be expressed.

To evaluate what kind of properties it is possible to express to qualify learning, we carried out a case study. Beyond the specificity of the considered application, it has been possible to elaborate general lessons.

The system is supposed to learn from its environment and/or from its users, but sometimes, conditions are changing, either occasionally or for a long time. It is essential to design the system in such a way that it can (1) detect those changes and (2) provide feedback when they occur.

This has two advantages. First, it is possible to use this feedback to express more accurate properties, and thus limit the number of false negative verdicts in the process of testing or monitoring. Second, it is possible to use this feedback to detect the inappropriate behaviour of the learning feature (for instance when training lasts too long or when the system always concludes that the conditions are changing).

Thus, designing a system with learning feature should not consist only in inserting an algorithm upon an existing system. One has to think about what kind of properties the new feature has to satisfy, after what the design should include the outputs necessary to evaluate them. This is necessary not only for the validation step but also to achieve transparency and accountability of the system [31]. This approach is called "design for testability" in software engineering [32].

The reader should note that providing feedback about learning achievement and environment evolution is something that goes beyond adding outputs. Learning features are often packed into independent modules that do not offer these outputs by default. So "design for testability" impacts the design of the learning features, before the system by itself.

Moreover, to judge the quality of a learning feature, it seems to us that safety and liveness property were a little bit too restrictive. We needed to be able to consider the evolution of different attributes during a period, from a statistical point of view, to capture suspicious behaviours. Typically, one could observe by this means the user satisfaction. As underlined in [16], the user might be the only possible oracle to judge the quality of the system outputs. If the user satisfaction is also collected, it will be possible to detect when it is mostly negative, and thus denote some learning troubles or inadequacy of the system w.r.t. the needs.

VI. Conclusion and perspectives

In this article, we report on a case study carried out to evaluate how a learning feature can be validated when embedded in a more global system. We chose to apply a monitoring approach, consisting in observing the outputs of the system during the execution to detect inconsistent behaviours. The difficulty was to express properties able to detect relevant failures related to the learning process.

The most important conclusion we have from this work is that the expected properties of the system should be considered during the design. While doing that, outputs necessary to evaluate those properties have to be included, otherwise, the final validation would be difficult. This impact the learning features by themselves, which, most of the time, provide results in a non-transparent way. Being able to provide feedback on the internal learning algorithm behaviour is the key to be able to validate the system, and to debug it if needed.

The work presented here has been carried out on a simple example. As perspective, we will validate other real systems that include learning feature, such as [33].

Notes and Comments: This work has been funded by the project CNRS-PICS 6999 and LIG emergence iCASATE project.

References

[1] M. deAgonia, "Apple's Face ID [The iPhone X's facial recognition tech] explained," *Computerworld*, 11 2017, https://www.computerworld.com/article/3235140/apple-ios/apples-face-id-the-iphone-xs-facial-recognition-tech-explained.html[Retrieved August, 2018].

[2] NewsDesk, "NTT develops AI system to match books with kids," *Asian News Network*, 23rd April 2018, http://annx.asianews.network/content/ntt-develops-ai-system-match-books-kids-71412 [Retrieved August, 2018].

[3] C. S. Smith, "Alexa and Siri Can Hear This Hidden Command. You Cant," *New York Times*, 10th May 2018, https://www.nytimes.com/2018/05/10/technology/alexa-siri-hidden-command-audio-attacks.html [Retrieved August, 2018].

[4] J. Wakefield, "Microsoft chatbot is taught to swear on Twitter," *BBC News*, 24th March 2016, https://www.bbc.com/news/technology-35890188 [Retrieved August, 2018].

[5] P. Lee, "Learning from Tays introduction," *Official Microsoft Blog*, 25th May 2017, https://blogs.microsoft.com/blog/2016/03/25/learning-tays-introduction/ [Retrieved August, 2018].

[6] D. Wakabayashi, "Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam," *New York Times*, 19th March 2018, https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html [Retrieved August, 2018].

[7] SOGETI, “Testing of Artificial Intelligence - AI Quality Engineering skills - An introduction,” Dec. 2017, https://www.sogeti.com/globalassets/global/downloads/reports/testing-of-artificial-intelligence_sogeti-report_11_12_2017-.pdf [Retrieved August, 2018].

[8] T. J. Lee, J. Gottschlich, N. Tatbul, E. Metcalf, and S. Zdonik, “Precision and recall for range-based anomaly detection,” *CoRR*, vol. abs/1801.03175, 2018.

[9] D. Mahajan, V. Gupta, S. S. Keerthi, S. Sellamanickam, S. Narayanamurthy, and R. Kidambi, “Efficient estimation of generalization error and bias-variance components of ensembles,” *CoRR*, vol. abs/1711.05482, 2017.

[10] S. Mukherjee, P. Niyogi, T. A. Poggio, and R. M. Rifkin, “Learning theory: stability is sufficient for generalization and necessary and sufficient for consistency of empirical risk minimization,” *Adv. Comput. Math.*, vol. 25, no. 1-3, pp. 161–193, 2006.

[11] C. Sammut and G. I. Webb, Eds., *Holdout Evaluation*. Boston, MA: Springer US, 2017, pp. 624–624.

[12] ——, *Cross-Validation*. Boston, MA: Springer US, 2017, pp. 306–306.

[13] A. Ramanathan, L. L. Pullum, F. Hussain, D. Chakrabarty, and S. K. Jha, “Integrating symbolic and statistical methods for testing intelligent systems: Applications to machine learning and computer vision,” in *DATE*. IEEE, 2016, pp. 786–791.

[14] Y. Tian, K. Pei, S. Jana, and B. Ray, “Deeptest: Automated testing of deep-neural-network-driven autonomous cars,” *CoRR*, vol. abs/1708.08559, 2017.

[15] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. L. Traon, “Empirical assessment of machine learning-based malware detectors for android - measuring the gap between in-the-lab and in-the-wild validation scenarios,” *Empirical Software Engineering*, vol. 21, no. 1, pp. 183–211, 2016.

[16] A. Groce, T. Kulesza, C. Zhang, S. Shamasunder, M. M. Burnett, W. Wong, S. Stumpf, S. Das, A. Shinsel, F. Bice, and K. McIntosh, “You are the only possible oracle: Effective test selection for end users of interactive machine learning systems,” *IEEE Trans. Software Eng.*, vol. 40, no. 3, pp. 307–323, 2014.

[17] R. Kurozumi, K. Tsuji, S. Ito, K. Sato, S. Fujisawa, and T. Yamamoto, “Experimental validation of an online adaptive and learning obstacle avoiding support system for the electric wheelchairs,” in *SMC*. IEEE, 2010, pp. 92–99.

[18] F. Doctor, H. Hagras, and V. Callaghan, “A fuzzy embedded agent-based approach for realizing ambient intelligence in intelligent inhabited environments,” *IEEE Trans. Systems, Man, and Cybernetics, Part A*, vol. 35, no. 1, pp. 55–65, 2005.

[19] K. Pei, Y. Cao, J. Yang, and S. Jana, “Towards practical verification of machine learning: The case of computer vision systems,” *CoRR*, vol. abs/1712.01785, 2017.

[20] R. Calinescu, C. Ghezzi, M. Z. Kwiatkowska, and R. Mirandola, “Self-adaptive software needs quantitative verification at runtime,” *Commun. ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[21] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu, “Properties of machine learning applications for use in metamorphic testing,” in *SEKE*. Knowledge Systems Institute Graduate School, 2008, pp. 867–872.

[22] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[23] L. du Bousquet, M. Nakamura, B. Yan, and H. Igaki, “Using formal methods to increase confidence in a home network system implementation: a case study,” *ISSE*, vol. 5, no. 3, pp. 181–196, 2009.

[24] G. Boracchi, M. P. Michaelides, and M. Roveri, “A cognitive monitoring system for detecting and isolating contaminants and faults in intelligent buildings,” *IEEE Trans. Systems, Man, and Cybernetics: Systems*, vol. 48, no. 3, pp. 433–447, 2018.

[25] S. Yerramalla, B. Cukic, M. Mladenovski, and E. Fuller, “Stability monitoring and analysis of learning in an adaptive system,” in *DSN*. IEEE Computer Society, 2005, pp. 70–79.

[26] V. Garousi, M. Felderer, and F. N. Kilicaslan, “What we know about software testability: a survey,” *CoRR*, vol. abs/1801.02201, 2018.

[27] R. Bache and M. Mullerburg, “Measures of testability as a basis for quality assurance,” *Software Engineering Journal*, vol. 5, no. 2, pp. 86–92, 1990.

[28] R. V. Binder, “Design for testability in object-oriented systems,” *Communications of the ACM*, vol. 37, no. 9, pp. 87–101, Sep. 1994.

[29] Institute of Electrical and Electronics Engineers, “IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries,” IEEE, New York, USA, Tech. Rep., 1990.

[30] A. Bertolino and L. Strigini, “On the use of testability measures for dependability assessment.” *IEEE Trans. Software Eng.*, vol. 22, no. 2, pp. 97–108, 1996.

[31] V. Dignum, “Accountability, responsibility, transparency - the ART of AI,” in *ICAART (1)*. SciTePress, 2018, p. 7.

[32] R. V. Binder, “Design for testability in object-oriented systems,” *Commun. ACM*, vol. 37, no. 9, pp. 87–101, 1994.

[33] K. Tamamizu, S. Sakakibara, S. Saiki, M. Nakamura, and K. Yasuda, “Capturing activities of daily living for elderly at home based on environment change and speech dialog,” in *8th International Conference Digital Human Modeling. Applications in Health, Safety, Ergonomics, and Risk Management: Health and Safety (DHM)*, ser. Lecture Notes in Computer Science, vol. 10287. Springer, 2017, pp. 183–194.