AmITest: a framework for semi-automated testing of Ambient Intelligence environments

Nikolaos Louloudakis

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science
University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Constantine Stephanidis*

This work has been supported by the Institute of Computer Science (ICS),
Foundation for Research and Technology Hellas (FORTH).

University Of Crete
Computer Science Department

AMITEST: A FRAMEWORK FOR SEMI-AUTOMATED TESTING OF AMBIENT INTELLIGENCE ENVIRONMENTS

Thesis submitted by
Nikolaos Louloudakis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____

Nikolaos Louloudakis, Department Of Computer Science

Committee approvals:_____

Constantine Stephanidis
Professor, Thesis Supervisor

_____

Anthony Savidis
Professor, Committee Member

_____

Margherita Antona
Principal Researcher of ICS-FORTH, Committee Member

Department approval:_____

Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, April 2017

I. TABLE OF CONTENTS

Contents

II. TABLE OF FIGURES

III. TABLE OF CODEBLOCKS

IV. ΠΕΡΙΛΗΨΗ

Η Διάχυτη Νοημοσύνη (Ambient Intelligence - AmI) έχει σταδιακά εξελιχθεί από ένα ερευνητικό αντικείμενο σε μία καθημερινή πραγματικότητα (π.χ. Internet of Things - αντικείμενα συνδεδεμένα στο internet), δημιουργώντας στους τελικούς χρήστες (είτε αυτοί είναι επαγγελματίες του IT ή χρήστες με ελάχιστες τεχνολογικές γνώσεις) την επιτακτική ανάγκη να μπορούν να προσαρμόσουν ή να προγραμματίσουν εκ νέου την συμπεριφορά Έξυπνων Περιβαλλόντων. Ο προγραμματισμός τέτοιων κατανεμημένων, ετερογενών και πολύπλοκων συστημάτων αποτελεί από μόνο του ένα δύσκολο έργο, πόσο μάλλον η διαδικασία επαλήθευσης της ορθότητας της συμπεριφοράς τους. Το τελευταίο μάλιστα είναι εξίσου δύσκολο αλλά ακόμα πιο σημαντικό, καθώς η επιβεβαίωση της ορθής απόκρισης ενός έξυπνου περιβάλλοντος σε συγκεκριμένα ερεθίσματα, αποτελεί έναν από τους πιο κρίσιμους παράγοντες για την αποδοχή του από τους τελικούς χρήστες.

Για αυτό το σκοπό, προτείνουμε μία υποδομή ελέγχου ορθότητας (testing) της συμπεριφοράς Έξυπνων Περιβαλλόντων, ονομαζόμενη AmITest. Η συγκεκριμένη υποδομή επιτρέπει με απλό τρόπο τον έλεγχο ορθότητας της συμπεριφοράς κάθε έξυπνου αντικειμένου ξεχωριστά, αλλά και του έξυπνου περιβάλλοντος ως σύνολο. Για να απλοποιήσει την συνολική διαδικασία ελέγχου, το AmITest, που αποτελεί σημαντικό κομμάτι του AmI Solertis (μίας εξειδικευμένης σουίτας προγραμματισμού περιβαλλόντων Διάχυτης Νοημοσύνης η οποία αναπτύχθηκε στο Εργαστήριο Αλληλεπίδρασης Ανθρώπου-Υπολογιστή του ΙΠ-ΙΤΕ), χρησιμοποιεί όλες τις μετα-πληροφορίες σχετικά με τα έξυπνα αντικείμενα που υπάρχουν στο περιβάλλον, καθώς και τους κανόνες της συμπεριφοράς τους. Πιο συγκεκριμένα, μέσω ενός απλού Wizard, ακόμα και οι λιγότερο τεχνολογικά έμπειροι χρήστες μπορούν να συνθέσουν εύκολα τους ελέγχους που επιθυμούν, ενώ η διαδικασία ενορχήστρωσης της εκτέλεσης των ελέγχων αναλαμβάνεται αυτόματα και εξ' ολοκλήρου από το AmITest.

Το AmlTest βασίζεται στην ευρέως διαδεδομένη γλώσσα προγραμματισμού JavaScript, σε συνδυασμό με έναν αριθμό υποδομών της, όπως οι Promises για ασύγχρονο προγραμματισμό και οι Mocha και Chai για την εφαρμογή ελέγχων (testing), στοχεύοντας την επίτευξη της μέγιστης αποδοτικότητας και της παροχής ενός αποτελεσματικού μηχανισμού ελέγχου ορθότητας με τον πιο άμεσο δυνατό τρόπο.

Το AmlTest παρέχει τη δυνατότητα πραγματοποίησης των ελέγχων σε ένα περιβάλλον προσομοίωσης, στο οποίο οι έλεγχοι ορθότητας μπορούν να εκτελεστούν «απομονωμένα», χωρίς να επηρεάζουν τις πραγματικές συσκευές του χώρου. Αυτό επιτυγχάνεται με μηχανισμούς εξομοίωσης αντικειμένων (π.χ., προσομοίωση του τρόπου αντίδρασης ενός αντικείμενου στα διάφορα ερεθίσματα) και «τοπική» εκτέλεση κώδικα συμπεριφοράς.

Με το πέρας ενός ή περισσοτέρων ελέγχων, το AmlTest μέσα από μια πλούσια γραφική διεπαφή παρουσίασης αναφορών, μεταφέρει τα αποτελέσματα εκτέλεσης τους στον χρήστη, ώστε να επαληθεύσει την ορθότητα ή μη της συμπεριφοράς του περιβάλλοντος. Επιπλέον, αυτή η πληροφορία παρέχεται και ως είσοδος στο κεντρικό σύστημα του AmI Solertis για περαιτέρω μελέτη με στόχο την εύρεση προβληματικών συμπεριφορών και τη διασφάλιση της σταθερότητας του περιβάλλοντος μέσω συστηματικού ελέγχου.

V. ABSTRACT

Ambient Intelligence (AmI) has gradually evolved from an emerging research paradigm to an everyday reality (e.g., Internet of Things) and, therefore, end-users – ranging from IT-experts to novice users – have a need to program or adapt the behavior of "Smart" Environments. Programming such highly-distributed, heterogeneous and complex systems is a challenging task, let alone validating their behavioral aspects. The latter is a rather difficult but crucial process, since such factors eventually determine their acceptance and perceived usability.

The AmITest framework facilitates testing and validation of the functionality of each individual artifact and of the environment's behavior as a whole in a straightforward manner. As a core component of the AmI Solertis - a development and orchestration studio for AmI Environments developed in the HCI Laboratory of ICS-FORTH – AmITest utilizes the available meta-information regarding the installed artifacts and behavior definitions, in order to simplify the overall testing process. More specifically, a wizard-like process enables even less-experienced users to easily compose the desired tests, while the test execution orchestration is automatically handled by the proposed framework.

AmITest heavily relies on the widely used JavaScript language and a number of reliable frameworks, including the Promises for asynchronous programming and Mocha and Chai for testing, aiming to achieve maximum efficiency, along with the provision of an effective validation mechanism in the most straightforward manner.

To minimize its footprint and increase its usability, AmITest optionally offers a sandboxed environment in which testing can be executed without affecting the actual system and its artifacts. Specifically, it provides artifact impersonation capabilities (e.g., mock how a smart artifact reacts to stimuli) and local behavioral script execution.

Finally, the internal reporting system of AmITest delivers the test execution outcome to the tester through a rich Graphical User Interface (GUI) in order to assess the effectiveness of the behavioral script and validate the desired assertions and invariants. That information is also fed back to the AmI Solertis system to be further

examined for potential erroneous behaviors and to ensure the environment's stability via its inherent continuous validation and integration process.

VI. ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor, head of the Human-Computer Interaction Laboratory of the ICS-FORTH and professor at the University of Crete, Dr. Constantine Stephanidis, for his trust, continuous support, overall supervision and guidance in the whole period of my Master's Degree studies.

I would also like to thank Mr. Asterios Leonidis for the amazing collaboration and his continuous support, valuable advices and assistance for this whole period.

I am also grateful to Dr. Anthony Savidis, professor of the University of Crete and principal researcher of the ICS-FORTH, and Dr. Margherita Antona, principal researcher of the ICS-FORTH for their participation in the supervisory committee.

In addition, I would like to thank all of my colleagues for our excellent collaboration during my Master's studies.

I would also like to thank all my friends for supporting me through this period. But most of all, I would like to thank my parents, Andriani and Konstantinos, but also my brother Stefanos, for their continuous support, love and encouragement, and for standing by my side through this whole period. Without them, I would not be the person I am today and they mean so much to me.

Last but not least, I am grateful and thankful to God, for giving me this amazing opportunity to learn new, amazing things, help others when possible, meet some amazing people, and become a better person through this whole process.

στην οικογένειά μου

1. INTRODUCTION

As Information Technology evolves, the traditional interaction paradigm where humans are just the operators of stationary machines is revolutionized by the concepts of Ambient Intelligence [1][2] and Pervasive Computing [3]. These concepts introduce innovative ecosystems in which humans are surrounded by ubiquitous technological artifacts (e.g., sensors, actuators, smart devices, etc.) and computational units that enrich their environment in a smart, transparent and unobtrusive way. In such environments (i.e., AmI Environments), ubiquitous artifacts can reason, collaborate and interact proactively in order to improve the quality of life of humans, by satisfying their needs and offering assistance in their daily activities.

The list of the application domains whose users could be benefited by such intelligent environments is rather endless, ranging from the automation of repetitive tasks (i.e., daily routines) in order to offer more free time to their inhabitants for other activities, to improving the overall quality of life of specific groups of people (e.g., people with disabilities, elderly, etc.) by assisting them with their daily activities.

A key requirement of an Ambient Intelligence ecosystem is that their behavior and interaction policies need to be easily programmed by their end-users, who most likely will not be computer professionals. A very important aspect though is that such environments are of high architectural and computational complexity. Therefore, not only programming is a challenging task, but also the proper testing and validation of the behavior of those environments is of utmost importance. Towards such objective, this thesis proposes the AmITest Framework, a testing framework for AmI environments.

AmITest aims eventually at supporting both the users that define the behavior of the Smart Environment (i.e., Smart Environment programmers), as well as the end-users of the Smart Environments themselves (e.g., house inhabitants, doctors, teachers, students etc.) with the least learning curve. The programming expertise of AmITest users however may vary greatly ranging from motivated users, who want to modify the intelligent environment they work into, to experienced IT professionals who determine in detail the behavior of the environment and the contained facilities. In order to accommodate the full range of IT skills of its users, AmITest aims to ultimately simplify the testing process, by providing features of guided test composition, along

with visual programming and other sophisticated programming capabilities for the tests definition. The AmITest framework is a novel part of the AmI Solertis Framework [4][5] that provides testing capabilities using well known testing and assertion frameworks in order to test the artifacts of the Smart Environment being tested as subject.

1.1. Problem Statement

Inside an Ambient Intelligence ecosystem, an increasing number of artifacts exist, work together and collaborate at the same time, with the ultimate aim of assisting the lives of its inhabitants, meeting their needs and making their lives easier in the most effective way. However, in order for the ecosystem to meet such objectives effectively, it must work properly and as expected at all times, without problems, and, even if a problem arises, it must be detected and resolved in the most optimal way, in the shortest possible amount of time. In the case that this is not met, the system might fail its aims in partial or fully, or, even worse, act against its inhabitants explicitly or implicitly, leading to catastrophic sequences to its inhabitants. Therefore, the Ambient Intelligence Environment needs to be tested thoroughly in an effective manner. In fact, the ecosystem must be in a continuous testing state while working, in order to ensure that it is working properly, and that any flaws detected while operating will be resolved as effectively and fast as possible and that, if this is not possible, the system is going to proceed to failsafe actions and inform its inhabitants about the cause and effects of a potential problem.

One state-of-the-art system for the programming of Ambient Intelligence ecosystems is AmI Solertis [4][5]. AmI Solertis is a complete studio for the design and orchestration of ambient intelligence systems that provides a complete set of tools for that cause, developed in the context of the ICS-FORTH Ambient Intelligence Research Programme. However, providing programming capabilities for such an ecosystem is not sufficient; as explained before, the validation of the system behavior is crucial.This thesis introduces AmITest, a complete and sophisticated, yet easy to use suite for the testing of the behavior of Ambient Intelligence applications, whose programming

behavior is specified in the AmI Solertis system. AmITest is a core module and an integral part of AmI Solertis, aiming to ensure that the behavioral programming of an AmI ecosystem is performed correctly and in an efficient way.
AmITest provides a complete set of tools to the developer of the Smart Environment, that the developer can use in order to validate the behavior of the system.
In AmI Solertis, the behavior of the Smart Environment is defined via a set of scripts deployed and executed. Such scripts handle the artifacts of the ecosystem as simple programming modules. Taking this into consideration, AmITest is able to test the behavior of those artifacts individually, but also test their behavior defined inside an AmI Solertis script. This procedure is independent of the deployment state of such a script, giving the ability of sandboxed execution of the script in order to check the validity of its behavior.

1.2. The AmI Paradigm

At the very beginning of modern Computer Science, computers were constructed for the performance of very specific tasks, mainly focusing on the part that computers know how to do well: fast computations. With computers large as half a football field, those machines focused on the performance of very specific tasks, forcing their users to conform to the strict requirements of their usage, such as printing cardboards, etc. This was used for many alternative usages, from flying to space till managing to beat the Chess World Champion Gary Kasparov at that time. Yet those tasks remained very specific and inside limited contexts.

With the invention of Personal Computers, computers started becoming more of toolsets under a more specific area that would play the role of assistants to the user. Then internet came to place, and the whole world started connecting via those "assistants", that provided access to limitless information. In parallel, mobile devices started gaining ground – and reducing in size, until we reached a new era: the era of *Smartphones*, mobile, handheld devices, primarily working as phones but also containing a number of useful features out of the box, including the active provision of access to the internet, attempting to meet the needs of their users.

Today, this “smartness” of devices has reached to a new level: devices increased their number of features, aiming to meet the needs of their users as effectively as possible. For that cause, the devices increased both their number of operations and their abilities to infer information from their context of use. Among other features, the devices started including the automatic usage and access of the internet in order to improve their features and ultimately attempt to cover the needs of their users optimally.. This resulted into the era of the *Internet of Things*[6], meaning is the inter-networking of physical devices, vehicles (also referred to as "connected devices" and "smart devices"), buildings, and other items—embedded with electronics, software, sensors, actuators, and network connectivity that enable these objects to collect and exchange data.

Nowadays, we have reached into talking about a new era; the era of *Ambient Intelligence*: instead of monolithic systems that heavily rely on the direct interaction of their users, we can talk about systems – in fact, whole environments consisting of both technological artifacts of diverse technologies and human inhabitants, in which these artifacts behave asynchronously, yet harmoniously between each other, sensitive to the presence of people and dependent on their direct, but also indirect interactions with them. Ultimately, the environment should aim in the optimal coverage of the needs of its human inhabitants, but also on act proactively regarding its healthy and effective operation, applying “critical thinking” and “smartness”, in a sense.
These artifacts eventually compose a unified computing entity, which is *pervasive*, *ubiquitous* and *unobtrusive* inside the human environment, yet it does not only exist, but actively acts for the benefit of its human inhabitants.

The term *Ambient Intelligence* was originally developed in the late 1990s by Eli Zelkha and his team at Palo Alto Ventures for the time frame 2010-2020 [7].

1.3. Programming in AmI

Some of the main characteristics of an AmI environment is that it consists of a set of diverse technological artifacts, each one having its own traits, and that those artifacts have to work in harmonious, yet asynchronous collaboration between each other, being first class citizens of a technological ecosystem, in order to cover the needs of its human inhabitants. The long-living optimal usage of those artifact systems is highly desired, considering that their acquisition contains important financial costs and that a potential malfunction could cause problematic behavior and result to additional time and financial costs regarding their repair or replacement. .

In order for the system to be programmable, all the above factors must be considered. An AmI ecosystem is a very complex entity consisting of a number of smaller, yet complex entities by themselves. All these entities communicate between each other asynchronously, and although they perform atomic, internal operations, they are implicitly or explicitly associated between each other.

In order to maximize the efficiency, extensibility and adaptation to the needs of their users, AmI systems need to be programmable, in fact not only by professional developers, but also by the users of the system itself, giving them the ability to program and modify the behavior of the system itself, based on their current needs, being able to modify it later on, or extend its behavior even further. Considering though that their users are primarily non-computer professionals, giving them the ability to program those environments is a task difficult by itself, as those environments are of high architectural and computational complexity. In addition, it is of high importance that those environments work as expected, making the testing and the validation of their behavioral aspects a crucial part of the development process.

In conventional systems, several cases require the provision of programming capabilities to non-computer professionals. This is performed mainly for reasons of learning the programming fundamentals via specific paradigms, such as jigsaw puzzles [8] and diagram connection mechanisms [9][10], each one having their own advantages and disadvantages [11]. Another area of usage of such provision is for the purposes of system configuration and tasks automation [12] but also the automation

of cause-effect operations. In that case, the user is able to specify specific actions performed (effect) when an event occurs (cause)[13]. In addition, there are systems that provide simulation-based behavior, such as the configuration of visual components inside a 3D model space [14].

In Ambient Intelligence environments, such paradigms can be partially introduced, along with other important aspects: considering that the actions are applied on an environment, end-user programming capabilities must be provided along with a number of information, including the particular sub-systems which exist inside the AmI environment, some descriptive information of how they work, along with the provision of related information about them. Paradigms such as jigsaw puzzles can be used effectively for small "scripts", considering that the programming of such systems must be straightforward, avoiding flooding the developer of the system with much information or requiring much configuration via large code snippets.

1.4. The Role Of Testing In Programming

As modern software size and number of technologies has come to an explosion since its birth – especially in the last 10 years. While in the first years that programming appeared, software code was limited to a few hundreds to thousands of Lines of Code (LOC), it was more manageable in terms of error detection mainly via static testing, and only the usage of very basic dynamic testing: code inspection and reviews, along with the usage of simple assertions, despite the lack of a variety of high level programming languages. Nowadays though, we have reached millions to billions LOC in large-scale projects, and this code cannot be easily managed for flaws and errors.

It is widely known, that "to err is human". Humans write code that will in most cases contain undesired behavior (we call them bugs), that can lead to the problematic or even catastrophic behavior of a system. While there are situations where minor bugs are manageable and tolerable, in general they are considered unwanted and need to be fixed as soon as possible. Furthermore, it is important to assist the developer create less such. A number of tools and techniques have been developed in order to assist the

developer avoid problems while writing code, from the syntax analysis and error detection that typical compilers provide, to sophisticated auto completion capabilities.

Eventually, the aim is to have a system with no problematic behavior. But in order to be sure about that, one has to check all the possible states of the system in order to ensure this is true. In practice, this is not possible. However, we can lead the system to the most common and important states, and check for the desired behavior. To this purpose, a system needs to be tested for the correctness of its behavior.

Nowadays, testing consists of a crucial procedure inevery project taken seriously. Therefore, the need of automatic inspection and analysis of code emerges.

In many cases, this procedure has also become automated; a number of tests are executed in various development phases. This process is called Continuous Integration (CI)[15].

1.5. Modern Software Testing Methods

Testing is split in two major categories: static and dynamic testing. Static testing is about attempting to detect errors inside the code by analyzing it without executing it. Popular methods are code inspection, walk through and reviews. Dynamic testing focuses on the analysis of code via its execution. Nowadays, there are three main categories regarding dynamic testing [16] [17]: Unit Testing, Integration Testing and End-to-End testing.

i. Unit Testing

Unit Testing focuses of the testing of the smallest code units inside the code. While the term “smallest unit” is not well-defined, the concept is to perform tests on single lines or small snippets of code that does not perform time-consuming operation such as network or file manipulation calls, in order to check its logical validity. Such tests are considered very fast, and are used extensively while performing Continuous Integration (CI) procedures, and provide better isolation in terms of the units under test, applying tests into small programming units, without including external operations such as disk IO or network operations.. Their results though provide less

confidence on the result of the testing process, because to their limited testing scope – the success of a specific unit test cannot provide much guarantees regarding the valid total operation of a system.

ii. Integration Testing

Integration or Service Testing aims to check the validity of the system by validating the behavior of its integral parts – that can be from functions to whole components. Such components may or may not include more complex operations, such as the usage of API calls via network, disc I/O operations and more. While Integration Tests check a larger portion of the system, they are much slower that Unit Tests, due to the application of time-consuming operations, such as those described above. Therefore, their usage is not as extended in numbers as Unit Tests in CI cycles. Integration Tests though provide better confidence on the test results of the systems, providing a better view of the total behavior of the system, as their testing scope goes beyond just a few lines of code, and focuses on the components, the "building blocks" of the system, along with their internal and/or external operations, thus providing less isolation than Unit Tests.

iii. End-To-End Testing

End-To-End or UI Testing relates to the testing of the system by performing operations as a user to it, usually at the front-end, and then validating its behavior. This commonly includes the automation of form filling, button clicking and navigation operations, aiming to test the behavior of a system as a whole, or, at least, split to its largest components. While such testing can provide valuable results regarding the overall behavior of the system, similar to how a user would use it, their application results to the usage of higher level parts of the system, without providing testing capabilities at low-level, internal mechanisms, thus providing limited flexibility and isolation of the parts of the system. In addition, as those tests use large parts of the system under test, they are much slower than integration tests. They can provide a total view of whether or not the main parts of the system behave well or not, giving big confidence of their testing outcomes, considering that if such tests pass, there is a very high possibility that the system behaves as expected.

While these techniques are widely used with success in conventional software development, they cannot easily transfer to Ambient Intelligence, due to the facts that an AmI ecosystem consists of a number of artifacts that act asynchronously, but in collaboration between each other via a Service-Oriented Architecture. Therefore, unit testing could not apply, as most of the functionality of the artifacts under test uses external operations such as the network. In addition, the user might interact with the system via direct, but also unintentional, indirect actions. Therefore, conventional end-to-end testing would not be sufficient, as many actions from the system do not come as a result with User Interfaces. On the other hand, Integration testing could be effective, considering that each artifact inside the ecosystem could be considered as an integral part of it. Of course, unit testing and end-to-end testing could be applied in specific cases inside an AmI environment – such as in Continuous Integration but also the interaction validation of specific artifacts as isolated components inside the ecosystem, but the main point is that the AmI ecosystem is practically an entity consisting of many smaller integral parts, that use external operations, thus making integration testing the most ideal candidate as the basis of testing in Ambient Intelligence.

1.6. AmI Solertis

i. Overview

As mentioned previously, an Ambient Intelligence (AmI) ecosystem, also known and as Smart Environment (SE) consists of a variety of different sub-systems or *artifacts* involving a considerable number of diverse technologies. These artifacts are not isolated entities though. In fact, they are considered first class citizens of the ecosystem. In an AmI environment, those "citizens" must have the ability to communicate between each other, or be aware of events that occur within the ecosystem, in order to proceed with actions depending on their logic in case they are affected. To this purpose, a protocol, or a common specification must be defined.

The aim of an AmI environment is to acknowledge the needs of its human inhabitants and to attempt to meet them in the best way possible. This means that the behavior of the ecosystem must be a response to the cues given from those inhabitants, in

combination with a number of side actions and results of the inference of ecosystem mechanisms.

In order for the artifacts of the Smart Environment to be able to behave in a way that will eventually meet the needs of the ecosystem's human inhabitants, their behavior must be adaptable, and, therefore programmable. There is a very important aspect about the programming of an AmI environment though; in traditional systems, their developers are usually people with expert IT skills, far beyond the skills of the median end-user of the system. In AmI environments, in order for the ecosystem to be able to behave the best possible way according to the needs of its end-user, the end-user should be able to heavily customize its behavior. This leads into the fact that the inhabitants of the ecosystems could be potential "developers" of the system, but without necessarily having professional programming knowledge – in fact, their Information Technology (IT) skills might differ, from totally novice to expert. Therefore, the ecosystem should provide programming capabilities in a transparent, yet simplified and thorough way, also considering and covering a wide range on human inhabitant knowledge levels and skills on IT. Considering also that the sub-systems of the AmI ecosystem behave as first class citizens of the ecosystem and that their behavior is affected from the behavior of other systems, their programming should not be performed individually, but as a part of a greater entity, a unity of those sub-systems. In practice, this means that the programming capabilities provided to the developer of the ecosystem should be unified. Considering though that the AmI ecosystem consists of a number of technologically diverse artifacts of high computational and architectural complexity leading into a large and complex ecosystem, the "developer" should follow a "divide and conquer" approach. In addition, the complexity of those sub-systems should be hidden from the developer for the sake of programming simplicity, including the complexity of their interactions.

In addition, the behavior of such an ecosystem needs to be validated at times, in order to ensure that not only the needs of its human inhabitants are met, but that the system does not behave against the sake of its human inhabitants. For that cause, a testing mechanism should be defined that will examine and test the behavior of the ecosystem and its sub-systems, in order to ensure their correct functionality and behavior.

All these facts and challenges lead to the need of a unified programming environment that will allow the programming of an AmI ecosystem as a whole, taking into account that the AmI ecosystem consists of a number of technologically diverse and complex sub-systems that communicate asynchronously. The system should hide such complexity as much as possible from the developer, and deal with the problems that arise from their asynchronous communication, while providing features for the straightforward and extensive validation of the ecosystem.. To this purpose, the *Ambient Intelligence Research Programme of ICS-FORTH* has developed *AmI Solertis* [4], a studio for Ambient Intelligence applications development.

In practice, AmI Solertis is a complete suite of tools that allow the management, programming, testing and monitoring of all the separate artifacts of an AmI Smart ecosystem, but also of the ecosystem itself as a whole. This is achieved via the usage of a number of different components, responsible for the collection, storage and analysis of metadata and information regarding the artifacts of an AmI Ecosystem, the generation of metadata regarding those artifacts, but also the provision of an Integrated Development Environment (IDE) that provides programming capabilities to developers with varying degrees of expertise and development skills, in order to enable the programmability of those artifacts. The aim is that the developers will be able to write simple scripts that will describe the behavior of the system, and then deploy and execute them, so that the ecosystem will follow the specified behavior. In addition, validation and monitoring mechanisms are provided, in order to ensure the proper functionality and the valid usage and business logic according to which those artifacts are used inside the ecosystem. In fact, the focus of this thesis is on the validation and monitoring components of AmI Solertis. Those validation and monitoring components come together into a core module of the AmI Solertis, named AmITest. AmITest will be the main focus of this thesis, and, therefore, be analyzed and described in detail onto the next chapters, focusing on the importance of this validation mechanism inside an AmI Environment.

ii. Motivating Scenario

The basic concept behind AmI Solertis is to make the orchestration and management of an Ambient Intelligence environment an easy and straightforward task. Ideally, one should be able to define the behavior of a Smart Environment just by using a few lines of code. For this aim to be achieved, a set of issues need to be taken into account:

- The artifacts existing inside the environment we want to orchestrate
- Their underlying technologies
- Interoperability and communication between artifacts and their environment,
- Guidance and orchestration of the systems
- Behavior of one artifact potentially affecting other artifacts
- Ensuring the correct behavior of such a system at all times

One can easily consider that this is an open list of issues. The key aspect of creating a Smart Environment orchestrator is to address such issues in a generic manner.

- Artifacts existing inside the environment

The artifacts that exist inside the environment should be identified in an automatic manner, or the developer of each should provide the required information about each.

- Underlying technologies

Upon Smart Environment orchestration, a mechanism will be used that will not be dependent on underlying technologies. As mentioned previously, each SE artifact could potentially consist of a number of different technologies. The Orchestration System should not care about this underlying technologies, but use a standard protocol for orchestration, that the artifact should be aware of.

- Interoperability and communication

All Smart Environment artifacts *should* be able to interoperate and interact between each other in a unified manner – in fact, they should be able to act on events occurring inside the Smart Environment at times, without even needing to know who caused them. Such mechanism can be applied via the usage of middleware technologies, such

as FAmINE [18], in combination with data and messages exchange protocols, primarily via the web,

- Guidance and orchestration

Considering that the environment might comprise a number of systems consisting of diverse technologies and that this number might increase as new artifacts can be added to the ecosystem, a standardized way of communication must be defined. In addition, in order to orchestrate their behavior, one needs to see the whole ecosystem following the paradigm of an musical orchestra: each of the musicians of the orchestra is a unique entity consisting of its own traits, musical background, knowledge and skills that can act individually, but ideally must work in perfect collaboration with the other entities, each one following his/her own musical score and being guided at the same time by the conductor who gives the musicians information regarding the music played real-time. Consider this metaphor applied to an AmI ecosystem, each artifact is like a musician, having its own traits, properties and "abilities", regarding what it can perform and what not. Each artifact though should be able to work in collaboration with other artifacts by following its own "score" along with the guidance of the "conductor".

As already mentioned, one of the main aims of the system is to provide programming capabilities on the artifacts. In practice, we could say that the "score" regarding each atomic artifact could be a set of commands that the artifact will follow asynchronously. This admission leads us to use a well-defined, and widely used programming paradigm that will support this kind of programming capabilities. To this end, the ecosystems defined by AmI Solertis follow the *Internet of Things* (IOT) paradigm. In fact, each artifact is considered as a component in the system, that follows the micro-Service Oriented Architecture paradigm (micro-SOA)[16], in terms that it contains its own dependencies that are decoupled from the other artifacts, yet it exposes its own *Application Programming Interface* (API) to the whole ecosystem. Furthermore, this API could be used from a programming script in order to define its behavior. In fact *AmI Solertis* follows this paradigm: each artifact exposes a *Representational State Transfer API* (REST API), a way to exchange information via the usage of a uniform and predefined set of stateless operations. This API is specified via a standard specification

for REST APIs, the *OpenAPI Swagger Specification* [19]. Using this specification, AmI Solertis is available to generate a proxy Module of JavaScript Programming Language [20], based on this API. This module can then be used in JavaScript programming scripts in order to define the behavior of the system is a thorough and straightforward way, using JavaScript mechanism to tackle arising problems – for instance, the usage of Promises[21] and async[22]/await[23] defined in ECMAScript2017 draft (ECMA-262)[24] for asynchronous programming. Those scripts are managed and finally executed from a number of AmI Solertis core components that constitute its Execution Runtime Environment (Solertis ERE). In addition, each artifact is sensitive to events occurring in the environment. For that cause, the Solertis ERE contains an event messaging system, based on the Redis PubSub[25] mechanism, responsible for informing interested artifacts for the occurrence of events inside the AmI ecosystem.

- Reciprocal effects of artifact behavior

In an Ambient Intelligence environment, all artifacts should operate as first class citizens of the environment; they act based on their atomic behavior, and respond to events occurring inside it.

There is no guarantee though that those artifacts are prevented from acting against each other; in fact, the behavior of one artifact could affect the behavior of another. This could result into negative consequences regarding the performance, proper functionality and behavior on the artifacts in particular, but also the entire ecosystem. In theory, this may lead to a practical race condition between two artifacts inside the ecosystem. But in practice, things are even worse: the behavior of an artifact could affect negatively multiple artifacts, and not only explicitly, but also implicitly, even between components that have no connections. Imagine that the behavior of an artifact A affects the performance of a component B. A third component C, dependent on the behavior of B but having no correlation with A, underperforms exactly because of the performance drop of B. This means that in practice, the behavior of A implicitly affects the behavior of C. Even worse, the word “underperforms” is generic here, and not well defined. In the end, the ability of connection between components could become very problematic, even catastrophic for the system itself and its inhabitants: what would happen if a Smart Environment regarding critical systems started to

underperform or malfunction? Of course, this is a situation that must be avoided at all costs, as we must ensure the proper functionality of the ecosystem.

- Ensuring the correct behavior of the system at all times

This is an open issue and a very important aspect inside Ambient Intelligence Environments. While such a valid behavior at all times is crucial, especially for life-critical systems, it becomes a losing battle fight, due to the complexity of an AmI ecosystem and the high level of dependencies, but also that this complexity grows exponentially as artifacts are added into the ecosystem.

In addition, one must consider that presence of the artifacts added inside the ecosystem can differ between stable and ad-hoc, making the task of behavior correctness validation even harder.

In practice, the proper behavior of the system at all times cannot be ensured, considering that the system consists of computationally complex entities that communicate asynchronously, thus in a complex way, between each other. What can be done though, is applying sophisticated, extensive, and repetitive testing onto the ecosystem from its design to its deployment, and attempt to tackle possible arising flaws as fast as possible. In addition, recovery and self-repairing mechanism could be useful in such situations. This increases the importance of the existence of a sophisticated testing component that will be able to validate the behavior of the system, from its design to execution time. This component should be able to face the AmI ecosystem artifacts as units, but also as micro-services, applying testing procedures to the maximum extent possible, providing basic capabilities of testing of the components of the system via simple constraints, to the mocking of existing artifacts of the system for testing purposes, or, even further, the usage of system logs for problematic functionality inference via a sophisticated problem detection engine.

AmI Solertis attempts to tackle this challenge through a core module of its suite, named AmITest. AmITest is a complete testing environment, regarding the validation of the behavior of the Smart Environment, by supporting a form of mostly integration testing of the behavioral and orchestration scripts of the system, in order to allow the validation of the correctness and the efficiency of the Ambient Intelligence system. This component will be analyzed in depth in the next chapters.

iii. System Architecture

AmI Solertis follows the principles of the micro-service architecture and consists of a number of smaller components, each one containing its own dependencies, without interfering with the dependencies of the other components. The purpose of applying micro-service architecture principles on the system is to provide flexibility and scalability: as an Ambient Intelligence system can scale up from one physical space to many, reaching to even larger systems such as smart cities[26], AmI Solertis should be able to scale up in analogy, in order to meet the needs of those systems.

AmI Solertis is separated into a number of smaller, yet autonomous components that are able to interoperate between each other, by exposing their own REST APIs between each other or even third parties, but hiding the large atomic complexity for the sake of simplicity and robustness. In order for this to be possible against a number of different components, they communicate via a common API specification. Considering that the exposure of those APIs is done via REST, we chose the OpenAPI Swagger Specification (OSS) as a solid, yet flexible way of describing those APIs. The usage of such specification allows each component to communicate between each other via REST APIs in a transparent way. In addition, OSS is used for the communication of Ambient Intelligence artifacts inside the smart ecosystem, so that they are able to communicate between each other and be programmable, as mentioned before.

In practice, the AmI Solertis system consists of components that are grouped into modules based on their functional role, and modules on their turn are grouped into categories. This module categorization does not isolate the system components but simplifies its model, making the system robust and scalable.
The AmI Solertis System is split into eight basic categories: *Universal Service Repository and Middleware, Packaging and Deployment, Execution and Health Monitors, Social Features, Universal Metadata Storage, Artificial Intelligence, Analytics* and *Authoring Tools*. Each of these categories consists of basic modules, necessary for the proper functionality of the system, providing a complete solution of features for the extensive analysis, deployment, programming and monitoring of the artifacts of an AmI ecosystem, and, therefore, the ecosystem as a whole.

As aforementioned, one of the main features of AmI Solertis and the focus of this thesis, is **AmITest [27]**, a component for the validation of behavior of the AmI Environment. AmITest is a complete testing suite and an integral part of AmI Solertis, included in the Authoring Tools Category, consisting of a number of internal back-end and front-end components that will be described further in the next chapters of this thesis.

Universal Service Repository and Middleware

Inside an Ambient intelligence system, each artifact has its own traits, properties, and behavior. AmI Solertis must know all these information in order to be able to use these artifacts. This module category consists of components mainly responsible for the information extraction, programmability activation, but also information management and service discovery. It consists of the following modules:

The API Extractor analyzes existing programming entities regarding the AmI ecosystem artifacts, extract valuable information, infer meta-information about the artifact and generate libraries and meta-components that enable programmability of the component, in a well-specified manner. In fact, the API extractor is responsible for the generation of a specification, named Swagger specification, a standard used for the definition of Service API, widely known and used as reliable.

The Proxy Generator uses a set of information collected regarding an artifact in order to enable easy and thorough programming capabilities on the artifact, hiding a large part of the complexity of an artifact, providing only the necessary parts to the developer of the AmI ecosystem.

The Service Back-End consists of a number of components that have the role of the manager of the AmI Solertis system. Those components are responsible for the proper metadata storage, the dependency definition, storage and management of each AmI artifact, but also the management of different artifact versions and their zero-configuration.

The Discovery tools acts as a repository for the services regarding the artifacts, as also a manager of information regarding the events occurring inside an AmI ecosystem.

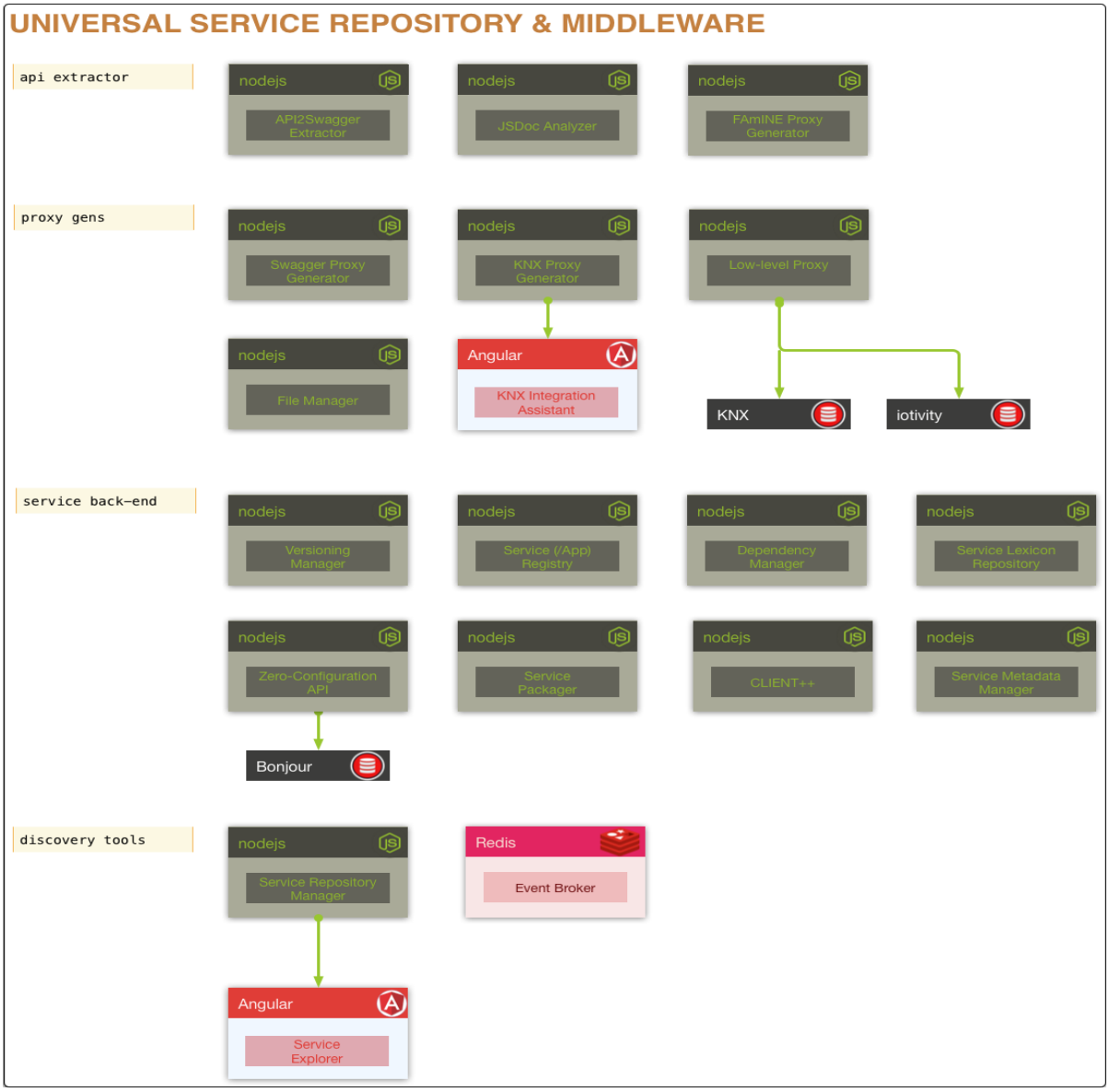

Figure 1- Universal Service Repository & Middleware Components

Packaging and Deployment

This module category consists of a number of components, related to dependencies and resources packaging but also services deployment and resolving in the AmI ecosystem.

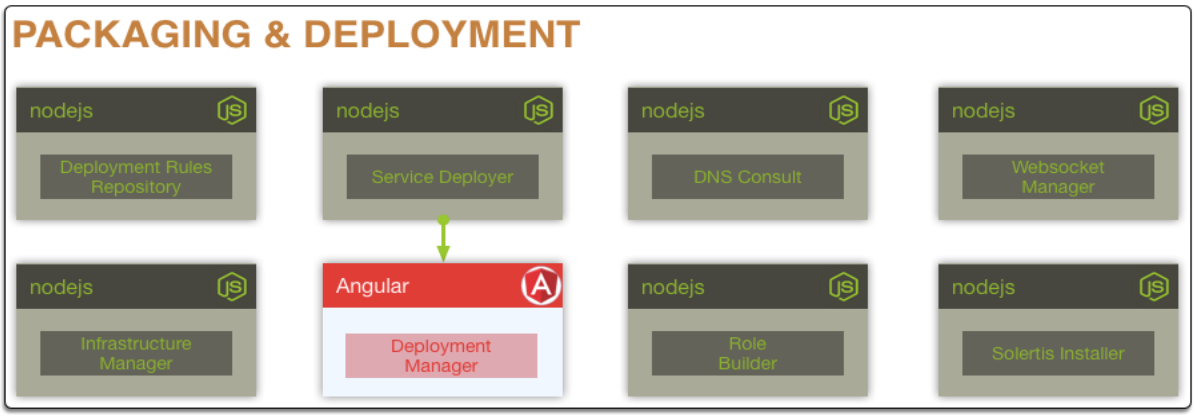

Figure 2 - Packaging and Deployment Components

Execution & Health Monitors

The category modules of this category are responsible for the deployment, execution and health status checking of behavioral and testing scripts of the ecosystem. It consists of the following modules:

The Error Checker consists of a number of components that allow the monitoring the executed behavioral and testing scripts and provide fault tolerance mechanisms related to the deployment and execution of the scripts.

The Logging consists of components responsible for the recording and the logging of important system actions, such as the events produced inside the ecosystem real-time.

The Real-Time Checker monitors the health of the ecosystem (scripts and services deployed, along with actions and events that occurred, etc) and proceeds to related actions if needed.

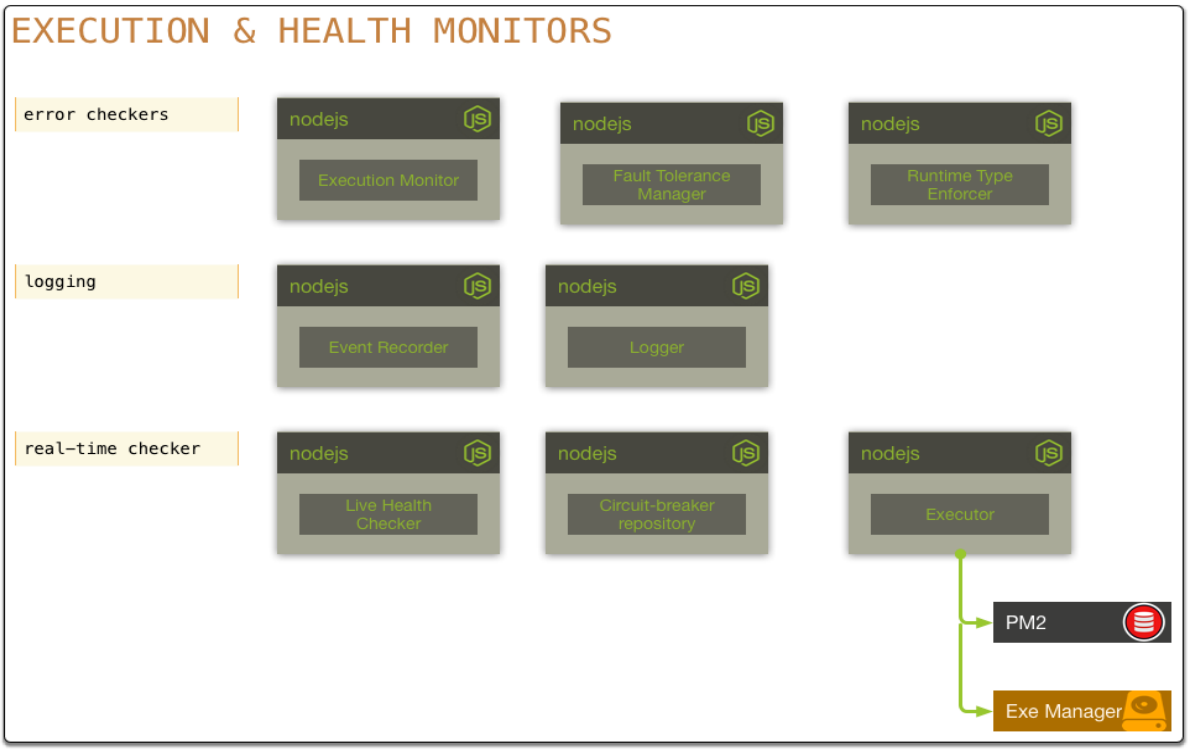

Figure 3- Execution & Health Monitoring Components

Social Features

This category consists of components related to the social and graph representation and profiling services of the AmI ecosystem.

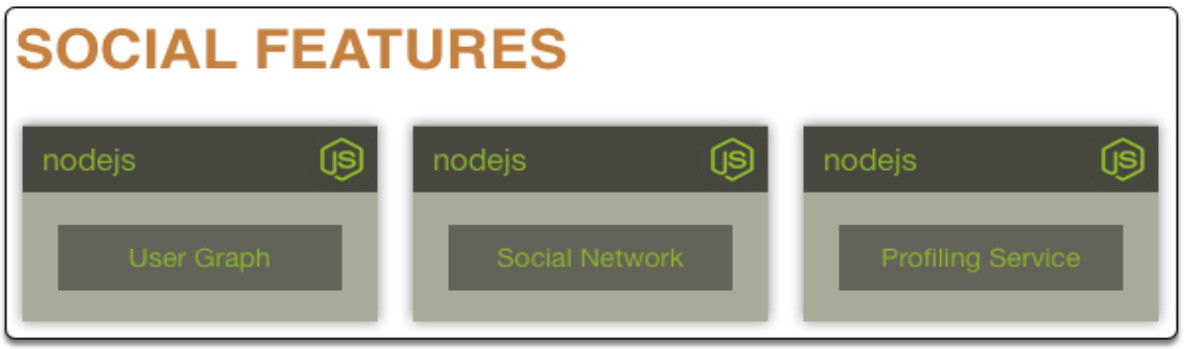

Figure 4 - Social Features Components

Various

This category consists of a number of components of generic purpose, such as the management of the universal documents and blobs regarding the AmI ecosystem.

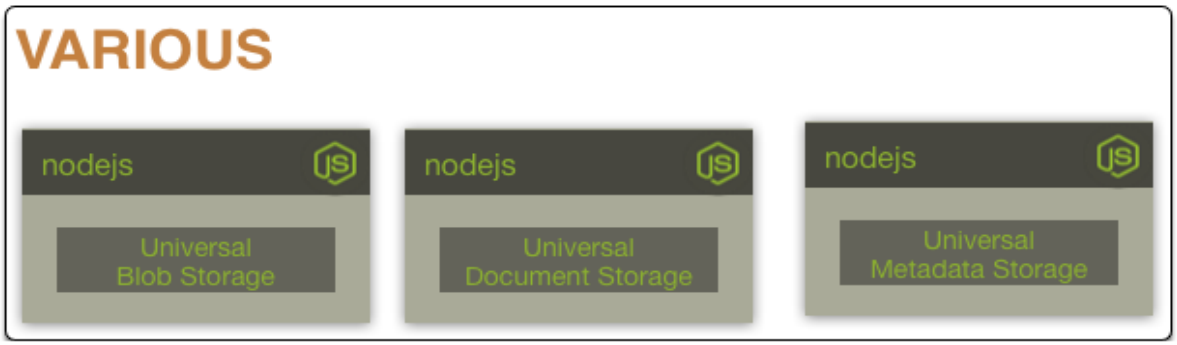

Figure 5- Various Components - Generic Purpose Category

Artificial Intelligence

In order for AmI Solertis to be able to provide the best programming solutions for an Ambient Intelligence environment, the system has to be able to infer semantics and information about the AmI environment and to be able to provide recommendations to its end-user. This means that the system must "reason" and "propose" solutions at times. It is also very important that the continuity of the AmI environment will be ensured. This category contains all the components that have to do with the Artificial Intelligence included in the system. The AI category consists of three main modules:

The Recommender which consists of components that infer semantic information from the artifacts, the resources, the dependencies and the scripts written inside the system, in order to give development options and to propose solutions to the system's end-user. As we mentioned before, this end-user can have from very little programming skills to be an IT expert. The system takes this factor into account when proposing solutions. In addition, the system contains components for the classification of artifacts and their APIs into categories.

The Handler module which consists of components related with the reasoning of degradation inside the AmI ecosystem, such as managing the execution flow in case an artifact fails to function properly.

The Smartness module which consists of components using metadata, logs and real-time data about the ecosystem in order to protect its long living and continuity and to recognize potential defects happening inside the system real-time.

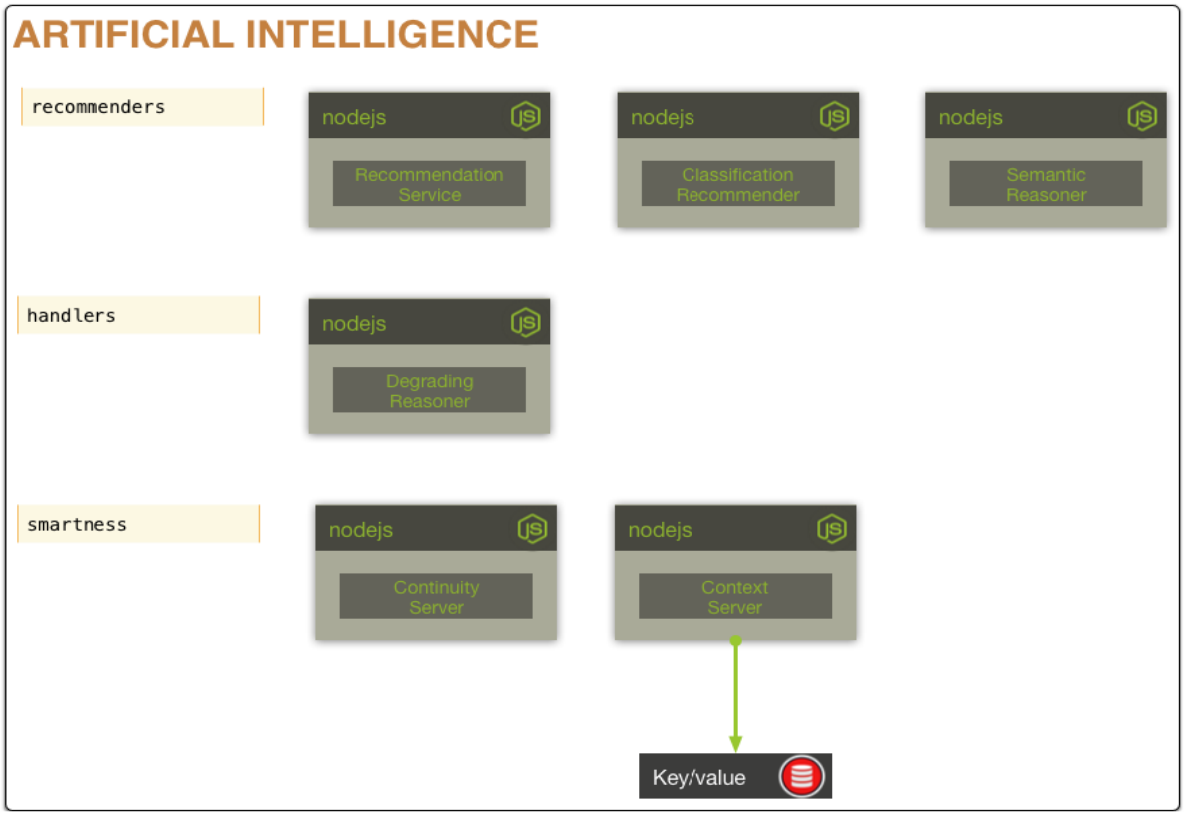

Figure 6 - Artificial Intelligence Category

Analytics

In order for AmI Solertis to ensure the proper functionality of the system but also be able to provide programming capabilities to users with a wide range of programming skills, the system must be in position of gathering data across the whole ecosystem in detail, analyze them, come to conclusions via inferencing and provide statistical information to the end user. This module category contains all the necessary modules in order to perform all the operations specified.

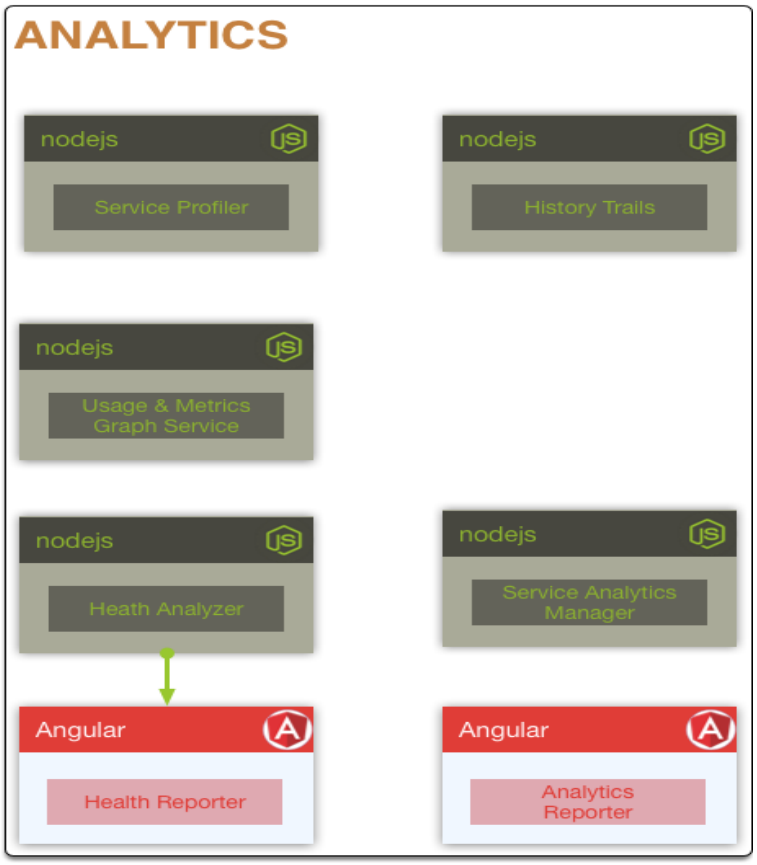

Figure 7 - Analytics Category

Authoring Tools

As mentioned above, one of the main aims of the system is to provide programming capabilities. Therefore, the system must provide a number of authoring tools that will assist the programming procedure, tools relevant to debugging, project management, code execution monitoring, integration and, most importantly, testing, provided via the AmITest framework, which is the main testing tool of AmI Solertis. As AmITest is the main subject of this thesis, it will be analyzed in the next chapters.

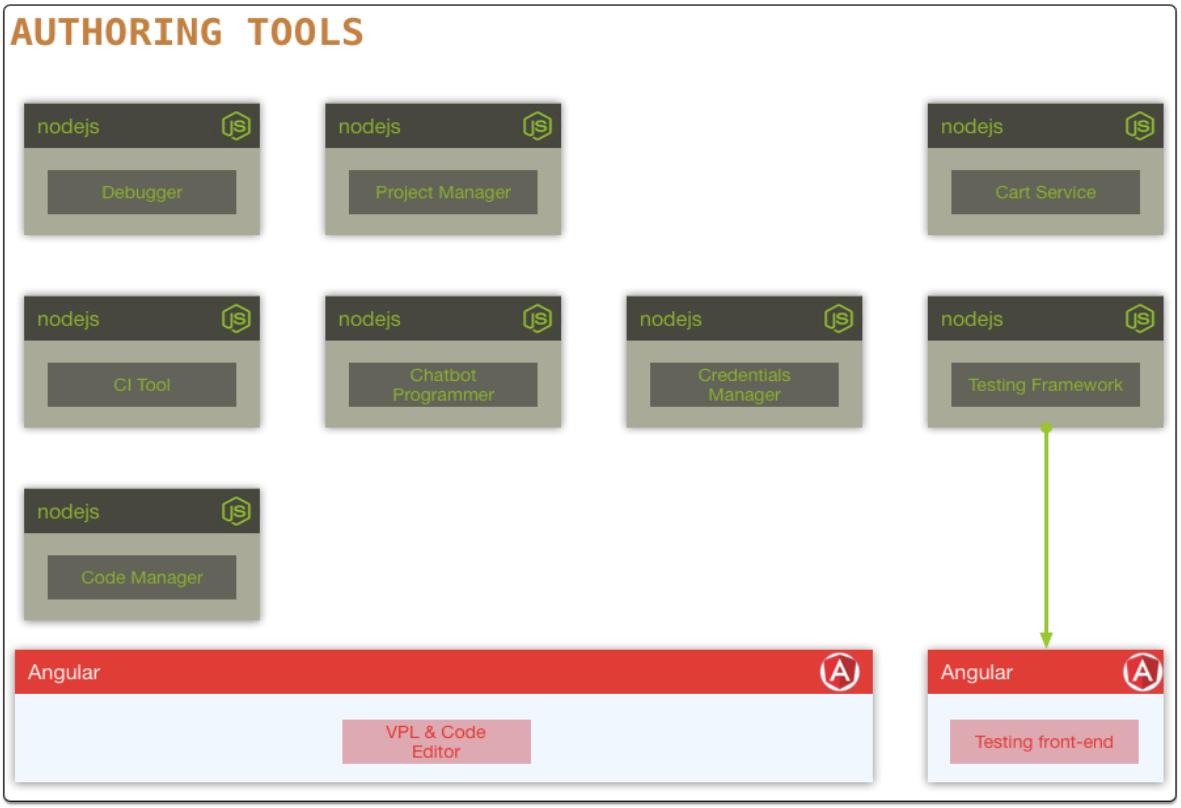

Figure 8- Authoring Tools Category

iv. The Need For Testing

Considering that an AmI Environment consists of a number of artifacts based on diverse technologies and that such devices are required to work optimally to the extent of their limits, but also to collaborate between each other in an asynchronous manner, it is of high importance to ensure that those systems will be proven reliable in their whole lifetime. To this end, the detection of flaws and problems in their usage must be easily and quickly performed, so that they can be replaced before they extensively apply their problematic behavior to the whole ecosystem, causing further problems. Furthermore, considering that the artifacts of the environment actively collaborate, any problems that could negatively affect that collaboration should be quickly detected and tackled.

In addition, an AmI environment acts as a "living organism", practically having its lifetime as a direct dependence of the lifetime of its components, but also having its

own “logic”, defined via a behavioral definition that the programmer of the Smart Environment has applied. In practice, AmI Solertis acts as an orchestrator of the AmI Environment, allowing the definition of its logic and ensuring its proper application inside the ecosystem. This logic, described in AmI Solertis via the usage of scripts defining the behavior of the environment, should be put under test in order to validate its correctness.

One must consider though that, applying extensive testing to the logic of the ecosystem, might lead to the extensive usage of the real technological artifacts existing into it, resulting to potential negative effects on their optimal functionality, practically causing damage to them and leading them to the limitation of their lifetime. Those artifacts though, are devices of a considerable cost; In fact, many devices might be of high technological cost – such as a Smart TV or a Smart Refrigerator inside a smart house.

Therefore, those artifacts should be protected from extensive usage – at least in terms of testing. The testing of the business logic of the ecosystem should be optionally decoupled from the usage of its technological artifacts.

The need of a validation mechanism becomes crucial. This validation mechanism should 1) validate the optimal functionality of the technological artifact existing inside the ecosystem and 2) validate the logic of the ecosystem, applied mainly by the behavioral scripts of AmI Solertis. In addition, this mechanism should apply testing and validation procedures on the business logic of the smart ecosystem by giving the ability of decoupling the testing of the business logic and the usage of the sub-systems at the same time.

As already mentioned, the programming of a Smart Environment is not an easy task, and therefore, testing and validation procedures are not easy tasks either. Considering that the programmers of the smart ecosystem can be actually its eventual developers, such validation mechanism should be available to users with a diversity of IT skills, hiding a large portion of complexity of both the behavior of the smart ecosystem but also the testing procedures applied, orchestrating automatically the whole process, making it as simple and straightforward as possible.

These needs led to the design and development of **AmITest[27]**. AmITest is a framework that provides testing capabilities to both the optimal functionality of the artifacts inside the AmI ecosystem, but also the business logic of the ecosystem described in AmI Solertis, while allowing the decoupling of such testing with the usage of the actual technological artifacts, via the provision of artifact simulation mechanisms (stubbing). In addition, the system aims to provide test composition capabilities to both IT experts and novice users, hiding a large portion of the complexity of both the AmI environment and the testing procedure, by guiding the user for the composition of tests and orchestrating their execution and reporting, providing eventually the results back to them via a rich Graphical User Interface (GUI), but also back to AmI Solertis for further evaluation. AmITest is the focus of this work, and we will discuss the system as a whole in the next chapter.

2. THE AMITEST FRAMEWORK

As we mentioned earlier, attempting to orchestrate an ambient intelligence environment, is not an easy task. The fact that a number of components consisting of a number of diverse technologies not only exist, but also heavily interoperate and rely part of their functionality on the proper functionality of each other, makes the control of the system literally a challenge. In addition, considering that the whole AmI environment ultimately aims to the coverage and support of the needs of its human inhabitants, the flaw of one or some of its sub-systems could lead into the partial malfunction or, if the flaw becomes extensive, even the collapse of the ecosystem, with annoying, the least, or even catastrophic consequences to its human inhabitants. This leads to the need of a comprehensive testing system that will be able to check for the validity of the ecosystem, report for relative flaws to the developers or even the inhabitants of the system and suggest solutions on the problems and, ultimately, attempt to fix potential flaws or enable catastrophe prevention actions. In order for this to be achieved, the behavior of the system must undergo dynamic testing, practically being executed and then evaluated and validated.

It is obvious though that the high level of dependencies between the system artifacts makes this a challenging task, especially in terms of the detection of errors and flaws.

In addition, an important fact that needs to be considered is that the developers of the ecosystem can simply be its inhabitants that might be IT experts, or even know very little from IT or programming. In order for the testing of the system to be effective, it must be a continuous, repetitive process, partially automated, but practically guided from the "developers" of the system which can be its inhabitants with very little IT knowledge and skills.

This thesis proposes **AmITest [27]**, a complete testing suite that provides a set of sophisticated tools that aim to test the validity of the behavior of an Ambient Intelligence environment thoroughly, but also supporting the usage of the system from both IT novice and expert users.

2.1. The Concept

In AmI Solertis, the usage of a number of scripts in combination with the real artifacts of the ecosystem, practically composes the behavioral definition of the ecosystem itself. In order to effectively test the system, one must focus on the ecosystem artifacts, their behavioral definition, and the combination of those. If we wanted to manually test the behavior of the system, we would set it up (artifacts activation and behavioral scripts deployment), then perform some actions and evaluate how the system would respond.

For instance, if we have a smart lamp and a motion detector inside the room, we could write a behavioral script that would define that if a user enters a room, the light activates. Its behavioral definition, in pseudo-code, would be something like this:

```
motionDetector = getDetector();
lamp = getLamp();
motionDetector.whenUserDetected -> lamp.setOn();
```

Code Block 1 - Simple Behavioral Definition (pseudo code)

Then we would deploy the script, get into the room, and observe if the lamp would light on.

In the testing framework, we would like to automate this process. We would like to consider the lamp and motion detector artifacts, deploy the script, yet simulate our entrance in the room and then check the status of the lamp, i.e., whether or not it was activated.

First of all, we would associate the test that we are going to compose with one or more behavioral scripts (such as the script written above), in order to put them under test. Then, we would select the artifacts that we would use, write the test code and execute it.

This particular testing would be very effective to be applied inside the actual

ecosystem, and could easily be achieved if three conditions are met: 1) the behavioral script is deployed in the AmI environment, 2) we have an action simulation mechanism (which in our case, simulates that the user entered the room), and 2) we are able to query the status of the artifacts (in our case, the lamp). We could then just write and execute such code, which, in a pseudo-code, would be like:

```
//gets the actual lamp
var lamp = getLamp();
//sets it to appropriate state
//before testing
Lamp.setOff();
//gets the actual motion detector
var motionDetector = getMotionDetector();
//simulates action behavior
motionDetector.simulateSomeoneWasMoved();
wait(2000); //wait some time
assert(lamp.IsOn());
```

Code Block 2 - A simple testing script (pseudo code)

Considering that the testing described above requires the execution of the code and is based on assertions, it is categorized on dynamic testing. While such testing behavior would be efficient in order to ensure that the artifacts of the system and the behavioral script work, there are cases where such test should not be applied in the actual environment. One simple cause is that, in practice we want to test the behavioral script *before* we deploy it into the ecosystem, in order to detect potential logical programming mistakes and fix them before its deployment, as a potential defect it could result into negative, or even catastrophic consequences for the ecosystem and its human inhabitants. In addition, we must consider that the AmI ecosystem consists of a number of artifacts, of which the acquisition is of a considerable financial cost, thus making their long-living optimal behavior highly desired.

To this purpose, in order for a particular behavior – or a number of behavioral script to be tested, a "sandboxed AmI environment" must be created, consisting of a number of scripts, artifacts (or some "impersonators" that will play their role), and a number of actions that will lead to some effects. The outcomes of those effects will then be validated. For that cause, the testing system must provide component impersonation (stubbing) capabilities, allowing both the testing of actual artifacts, but also give the ability to the tester to define and use such stubbing components, that will play the role of the actual artifacts in the testing procedure.
In simple terms, the stub in AmITest is a script that aims to behave in a way that will impersonate the behavior of a real artifact, in a very specific way. The aim of the usage of stubs is to be able to isolate particular artifacts of the system by stubbing others in order to test them, in an integration testing process [16]. In addition, stubs give us the ability to validate the behavior of the script in a "sandboxed" environment, i.e., without letting it affect the real AmI ecosystem, allowing to either test it before deploying it, or even isolating it from the real ecosystem for the sake of the validity check.

Of course, we still want to be able to test the behavior and the artifacts of the actual ecosystem along with the sandboxed one, providing validation capabilities in both cases.

AmITest uses a common assertion mechanism in both cases along with simple artifact management, allowing the definition of stubs and their direct usage without affecting the actual ecosystem, but also their enabling and disabling with just a "click". In fact, whether the testing system uses an artifact as a stub or as a real artifact, physical or programming, does not affect neither the behavioral script nor the testing validation process, allowing its definition via simple configuration and hiding the complexity from the tester.
Of course, another important aspect is the validation assertions definition. Whether or not an artifact is stubbed or a behavioral script under test is already deployed on the system or not, a number of validation assertions must be applied; these assertions will give a number of outcomes, which will provide important indications about the status of the artifacts and the implicit actions of the behavioral scripts inside the ecosystem itself.

AmITest provides a simple and effective assertion mechanism. Considering that it is a core part of AmI Solertis, its testing scripts are based on JavaScript code modules and popular JavaScript testing and assertion frameworks. In addition, AmITest provides a powerful reporting mechanism, in order to provide the outcomes of those validation assertion process to both its users (the tests) and AmI Solertis itself, for further analysis for automatic detection of problematic behaviors and Continuous Integration (CI) purposes.

2.2. Related Work

Considering that Ambient Intelligence is a paradigm with only a few years of active research and development, there is not plenty of research work done on the field of testing in literature. In fact, the majority of existing approaches mainly focus on the validation of design of Smart Environment. To the best of our knowledge, we did not find any work to involve end-users and system inhabitants, mainly focusing on the design process.

Regarding theoretical work, Heng Lu[28] proposes the application of testing inside pervasive environments based on the context of their computing entities, and to redesign the test cases including the context.

Ichiro Satoh [29] started first, back at 2003, working the AmI Concept on Mobile, Ad-hoc, “pervasive” Devices. Since there were no smartphones back at that time, Satoh attempted to approach the AmI paradigm via the usage of ad-hoc devices available at that time, such as PDAs. Those devices were extremely limited in terms of resources such as memory, but also available sensors and features provided. Satoh attempted to test the behavior of such ad-hoc devices by emulating their physical mobility via agents installed in various workstations inside a Smart Environment, testing network-dependency, mobility and multicasting –based management. Satoh mainly focused on the correct transition of those devices from place to place inside the environment. We must consider though that today’s available technologies of such process are

standardized via zero-based configuration and the use of standards such as Bluetooth reduce the necessity of such testing inside a Smart Environment.

UBIWISE[30] also back in 2003 is a system that simulates the existence of artifacts and their behavior inside a Smart Environment, by representing them in a Quake III Arena Graphics Engine 3D model, focusing on the simulation of scenarios regarding the extension of a system or the testing of its robustness without affecting the actual system.

DiaSim[31] is a simulator for pervasive applications, focusing on the simulation of a pervasive environment via a configurable editor aiming on the execution of simulation scenarios.

Similarly, UbiREAL[14] is a system that aims to provide simulation capabilities for a smart ecosystem, allowing the definition of the visualization of smart objects inside a 3D model which represents the AmI environment, along with their behavioral code, mainly focusing on the design process of a Smart Environment, featuring prototyping and realistic behavior simulation mechanisms.. Those objects can either be simulations of actual objects, or, via Universal Plug and Play (UPNP) [32], map to actual devices.

O 'Neil et al propose InSitu [33], a system focused mainly on situational testing, practically isolating states inside the simulation of a smart ecosystem and applying reasoning and generic rule checking in order to detect specific states inside the system, relying on a specific query language while allowing 3D virtual representation of the AmI environment. While such an approach allows the effective identification of flaws inside a virtual ecosystem via pattern matching on specific situations occurring inside the virtual system, it does not allow the testing on a real environment, focusing mainly on the design process via iterative prototyping circles, without giving capabilities of testing the real AmI environment. In addition, the system does not focus on the detection of flaws regarding the usage of the artifacts inside the environment.

In general, testing is not an easy task, and often requires a number of assumptions in order to be successful. DP-TraIN [34] is a Ambient Intelligence drug traceability infrastructure, , which supports end-user service provisioning in order to enable the pharmacy staff to create and execute their own services for facilitating drug

management and dispensing. The system was tested in the pharmacy department of Gregorio Marañón Hospital in Madrid, and, although it addressed a number of tasks of its users in general, assisting to reduction of time required for their completion, the system failed to provide an effective testing mechanism of the procedures, due to the lack of the existence of a sandboxed environment for the testing of services with example data before its deployment. AmITest attempts to approach the testing procedure by using existing metadata for the system itself, avoiding assumptions but analyzing and using actual system data in order to provide, among others, the capability of a sandboxed environment for the execution of tests regarding the behavior of the AmI ecosystem.

Regarding the design process, DAI Virtual Lab[35] is an environment for testing the usage of Ambient Intelligence applications by providing a visual 3D representation of the actual AmI environment via a 3D model, allowing the users of the system to apply tests on the environment without interacting with the actual environment. Such testing capabilities are considered very interesting in terms of giving the ability to the end-user to test the environment he/she uses, yet it does not provide a complete mechanism for testing the behavior defined inside the ecosystem and the correct behavior of the artifacts of the ecosystem, mostly focusing on the visual representation of the space and not on the extensive testing of any behavioral definitions inside the smart system. In addition, the system does not provide any testing mechanism regarding the actual environment, but only interactions with its virtual representation.

AmI environments often follow the paradigm of Service-Oriented Architecture (SOA). In terms of services definition, there are some tools, such as TASSA [36] that allow the testing of the definition of services defined in Service Models, such as BPEL. These testing tools provide testing capabilities of very limited scope, focusing on the design and the performance of specific services individually, but not allowing the extensive testing of an environment as a whole.. In addition, in order to test such services, expert IT users are required to conduct a complex testing process.

An Ambient Intelligence system consists of users that should be able to program their environment with the purpose of covering their needs, and that those users can be IT experts, but also novice users. In order for this to be achieved, other methods than

plain coding needs to be explored in order to enable programmability for novice user. In this context, it is worth mentioning some research and industry work done regarding the programming capabilities given for end-users. Considering that those novice users have little or no programming experience, other paradigms than coding should be followed mainly focused on composition interfaces, such as the Visual Programming paradigm [37].

Dahl et al [11] have applied some evaluation on the best practices regarding Visual Programming via composition interfaces. They examined the usability of three types of visual programming interfaces: filtered lists, wiring diagrams and jigsaw puzzles. From their results it emerges that filtered lists are a highly efficient way of specifying compositions, giving more confidence to the users, while wiring diagrams are not intuitive for the specification of a control flow, but were more useful for the representation of data exchange between entities. Finally, jigsaw puzzles are a more fun and intuitive way of building compositions, yet increasingly becoming more difficult to handle in terms of information extraction, as their size increases.

Inside an AmI environment, its human inhabitants – mainly non-computer professionals will mainly be asked to program and test its behavior in order to cover their needs, thus mechanisms that facilitate programming by such users with little or no experience are supplied, including visual tools that can be easily learnt and used in order to design programs and validation tests. This paradigm is effectively used in order to provide programming capabilities to systems that target non-professional users in various systems with diverse objectives. AmI Solertis enables the definition of the behavior of Smart Environments via visual programming even by novice users (e.g., house inhabitants or teachers). Scratch [38] is a visual programming environment primarily targeted to users of ages between 8 and 16 years, with limited to no programming experience, which aims to teach them programming while working on meaningful projects such as animated stories and games via a visual programming editor. Virtuoso [39] is a visual tool for creating educational games aiming primarily at non-professional users, based on Valve's game engine. TouchDevelop [40] is a system for developing applications directly from a mobile device through the cloud using a custom visual editor that adapts its functionality based on the knowledge and

programming skills of its user. App Inventor [41] is a platform from MIT which provides a web-based visual programming tool for designing mobile applications online, using Blockly[42], a block-based editor developed by Google. Microsoft's Visual Programming Language (VPL) [9] uses wiring diagrams in order to provide programming capabilities to end-users. Automator [43] is a visual scheduling tool providing capabilities of repetitive automation tasks in the Mac OSX platform. Zapier [12] is an online service for the automation of tasks between web apps. IFTTT [13] is another widely known online service that provides cause-effect definition capabilities for specific events and application services via a simple interface.

All the aforementioned systems facilitate programming of various kinds to users with very little or no programming experience via employing the visual programming paradigm. However, they do not provide testing capabilities. AmITest targets AmI environments where common testing techniques (i.e., Unit Testing) may not suffice as most of them lack the necessary testing and validation mechanisms to allow the verification of the behavior of the programs by their end-users.

AmITest aims to address those pitfalls as it not only supports testing of the behavior of a Smart Environment, but also offers both visual and script editing facilities to accommodate users with different levels of expertise. The aim is for any user will be able to program her own test cases and test the behavior of the Smart Environment easily. This could be achieved via either providing tools that would either assist the script composition process from IT expert users such as an editor with code highlighting and autocompletion capabilities, or the usage of existing visual programming tools such as Jigsaw puzzles and wiring diagrams, but also the provision of 3D interfaces for the orchestration of the environment without the usage of code from the novice end-users. Using these features, an expert programmer would be actively assisted in order to write a script that defines a behavior inside the ecosystem. In addition, a novice user, such as a teacher inside a learning environment would be able to specify parts of the behavior of its Smart Environment without the need of writing actual code, as the behavior of the ecosystem could be defined using visual programming. Text-based scripting support for end-user programmers is inspired by many well-established incarnations in the domain of electronic games development, with languages such as Lua [44] and JavaScript [20] having played an important role in

the widespread adoption of extensible game engines (such as Unity [45]), and even further, to the introduction of games that players can freely customize (e.g., the game "Second Life" offered the Linden Scripting Language [46] through which players were able to create in-game elements).

2.3. Testing Approach

AmITest is a framework that considers, but also differentiates from all the testing approaches described in the technologies and systems mentioned above, regarding its testing approach.

AmITest focuses on the creation of tests that aim to validate the proper behavior of the artifacts as integral parts of the system, similar to the logic of integration testing mentioned above, aiming to test the appropriate functionality of each artifact and its behavior as a unit inside the actual ecosystem by applying tests on them. AmITest also allows the execution of behavioral definitions of the AmI environment either inside the actual environment, or, optionally, outside it, within a "sandboxed" AmI environment which simulates the actual environment in a simple, yet straightforward way, via the impersonation of the actual artifacts of the ecosystem, but also the deployment and execution of the behavioral definitions of the actual system, allowing its tester to validate an AmI behavioral definition in isolation from the actual environment. In addition, AmITest provides a mechanism for the simulation of events inside the Smart Environment, as a part of the artifacts simulation mechanism. In terms of complexity of its use, AmITest simplifies the test composition procedure via a step-by-step wizard, while at the same time, it provides novice and expert end-user programming capabilities via a jigsaw-based puzzle editor (Blockly) and an online JavaScript editor (CodeMirror[47]) with auto completion, code highlighting and hinting capabilities. Finally, AmITest automates the test execution process, orchestrating the proper sandboxed environment initializations and the tests execution, but also the reporting process.

AmITest is a complete suite, provided as a module of the AmI Solertis system. AmITest is based on similar testing frameworks for untyped languages, and in particular on

testing and assertion frameworks for the JavaScript programming language, such as Mocha[48], ShouldJS[49] and Chai [50], but also Jasmine [51]. However the list is not exhaustive, as AmITest is extensible to more testing and assertion frameworks.

2.4. System Requirements

In order for AmITest to be able to effectively apply testing procedures in a complex environment such as an AmI ecosystem, it must be able to hide large portions of complexity from the tester, while automatically being able to orchestrate the whole testing process.

Considering that Smart Environments are complex systems with a considerable number of collaborating artifacts composing them, it is necessary to validate the behavior of each artifact individually, but also the Smart Environment behavior as a whole.

In terms of usage, the aim is to use a number of standardized, lightweight technologies in order to make the system less costly. In general, this procedure is on the time scale of minutes, and can be easily automated, performing setup operations by itself (such as the initial communication with AmI Solertis components), on initialization.

In terms of use, the aim of AmITest is to be used in the most simple, yet effective possible way. Considering that web applications are an emerging trend today and that internet connections have increased in speed and bandwidth, while reducing in financial cost of use, it would be very helpful if the tester could use such an application in order to apply his/her tests. This also gives the possibility to the tester to execute these tests from anywhere, without further requirements or resources than having a device with an internet browser. Even further, if the GUI of the application is mobile friendly, this could be applied with ease from mobile and tablet devices.

AmiTest provides an important set of features, mainly focusing on the provision of artifact impersonation and assertion mechanisms, regarding the AmI environment, in a clean, concise and straightforward way to the user. More specifically, AmITest provides the following features:

- **Test composition using an online, responsive application**, one needs only a browser in order to use the system. In fact, the plan is fro AmITest to be optimized for usage via mobile devices.

- **Usage of a dynamic, well-known, powerful scripting language (JavaScript),** which supports state-of-the-art mechanisms for asynchronous programming just by a few lines of code, such as Promises, and async/await operations.

- **Integration Testing via Assertions on Asynchronous code**, based but not limited to the use of a widely-used testing framework in JavaScript called Mocha, in combination with assertion frameworks, such as ShouldJS and Chai, providing APIs for unit and integration testing following a syntax logic similar to natural language, which, in combination with the asynchronous programming mechanisms in JavaScript, allow the composition of tests with just a few lines of code (or blocks, in case of Visual Programming).

- **Powerful stubbing mechanism** on the artifacts of the AmI ecosystem, based on the Service Oriented Architecture. More specifically, AmITest provides the ability of the creation of stubs that will be set up, initialized and used as services, containing their own context, while having lifetime equal to the duration of the test execution process.

- **Simulation of events inside the actual or the sandboxed ecosystem** via the extension of the APIs of stubbed devices that allow the registration of events within the ecosystem.

- **Rich and responsive Graphical User Interface reporting environment,** easily accessible from both desktop and mobile devices.

- **Provision of guidance mechanism in the test composition process**, aiming to reduce the possibility of errors.

- **Responsive Design Graphical User Interface**, providing abilities of test composition, execution and reporting via both desktop and mobile devices.

- **Extensive test code editing capabilities** via a very powerful JavaScript code editor, featuring effective and dynamically extensible auto completion and syntax highlighting.

- **Hiding of a large portion of testing orchestration complexity**, providing the tester with only the necessary information, while taking care of all the details and execution steps needed in order to prepare the test execution runtime properly, execute the tests and generate the related reports. This includes the generation of code
- **Basic Provision of End-User Testing via Visual Programming**, using a simple and straightforward test composition step-by-step wizard which manages the artifacts in lists, in combination with the Blockly editor for tests composition using jigsaw puzzles.

3. FRAMEWORK ARCHITECTURE

Being an integral part of the AmI Solertis studio, the AmITest framework follows the same architectural paradigm the whole studio follows: micro-service architecture. In fact, AmITest consists of a number of standalone and extensible back-end and front-end components that altogether collaborate in order to provide a sophisticated testing mechanism with extensive capabilities regarding the testing of Smart Environments..

In practice, AmITest aims to test the behavior of the system, by following the behavioral specification model that AmI Solertis follows: each artifact is eventually represented into the system as a JavaScript module, generated via the usage of its Swagger Specification automatically, upon artifact registration in AmI Solertis Studio. This module is used in order to put the specific artifact under test. This is done by using its API proxy module to modify its state in real time, or create a stubbing service that will play its role in the test process, especially for the testing of other real components, or in order to test the behavioral validity and efficiency of a behavioral script. Such stub services are NodeJS/ExpressJS services, practically impersonating the API of the real artifact. Considering that AmI Solertis defines the behavior of Smart Environments via the usage of behavioral scripts, AmITest can test the behavior of the system by validating the behavior of those scripts. And this the testing basis of testing in AmITest; the testing suite collects all the information it needs in order to test those scripts extensively and by testing them, it tests the behavior of the Smart Environment in practice. Of course, this is the basis for the testing of the behavior of the system, but it is probably not enough by itself in order to test the system extensively and effectively. AmITest needs mechanisms for recording and replaying situations inside the AmI Environment, but also mechanisms for continuous integration and behavioral flaws analysis and inference, along with user monitoring capabilities and problem recovery mechanisms. Considering that AmITest is a research work in progress, the system attempts to examine the effectiveness of the combination of unit and integration testing in AmI environments in its current state, by using mechanisms such as assertions and components mocking. Also, the system attempts to explore the

efficiency of flaw discovery and behavioral validity check via the recording and replaying of states of the system. More mechanisms are considered to be added to the system as a future work. Considering that the system is designed with the aim of being extensible, we expect even better results of its usage in the close future.

AmITest functionality is based on several major components: The main components of the system are *the Test Composition Wizard, the Test Code Generator, the Execution Runtime, the Metadata Information Extractor, the Files and Databases Manager and the Test Reporter*. Each of those components might consist of other smaller sub-components, especially in cases of handling a number of tasks, such as the Execution Runtime. In addition, AmITest uses data generated from other components, such as the API extractor and the Proxy Generators of the *Universal Service Repository and Middleware* category. Those components consist of smaller components that manage a number of important operations inside the testing system. This components aim to achieve the following aims: First of all they attempt to provide a complete Integrated Development Environment (IDE), in which the user of the testing system will be able to prepare tests with ease, following a standard, straightforward, step-by-step procedure. In addition, the environment works has been designed taking into account that those users might have different IT skill levels, from very novice to expert, and aims to be effectively usable by them, to the best extent given their skills. Additionally, the components mentioned above attempt to simplify the process of testing, by hiding complex operations from the developer and performing them in a standardized and automatic way, without the necessity his/her intervention. Those operations might be crucial but can be extremely complex even for an IT expert, such as the dynamic code generation of components needed inside the system or setting up a mocking environment for the sandboxed execution of a behavioral script, or managing the storage and loading of testing script metadata, resources and dependencies. Finally, the system attempts to give a fully detailed view of the behavioral health status of the system, and, to propose solutions if any defects are detected. In future versions, we plan extend this functionality by exploring more options on the system's self repairing, management and deployment. In fact, AmI Solertis itself, and, as a logical consequence, AmITest, can become artifacts of an Ambient Intelligence ecosystem, having automatic

control over their own behavior, if programmed that way. In the next sections, we will analyze the components described above, which constitute the heart of AmITest.

Considering that AmI Solertis follows the micro-Service Oriented Architecture paradigm and that AmITest is one of its core components, following the same paradigm for it was a logical design decision. As shown above, each of the components of the system is logically separated from each other, handling their own dependencies, while interoperating at the same time. In addition, sub-components are grouped into greater entities based on their functional role.

In general, the system follows a "divide and conquer" approach: each large component is split into smaller components, each one performing a small, very specific, yet crucial task, while being autonomous in terms of resources and dependencies. That way, the system becomes more robust and scalable, as each of its components can be easily modified internally without affecting the whole system. In addition, more components can be added with ease, adding extended functionality and additional testing capabilities to the system.

The User Interface design of the system was based on the basic, state-of-the-art usability principles. We attempted to use the most appropriate fonts, regarding to the usage of the system as a web application. We also followed a minimalistic design, along with the grid presentation of components wherever possible. In addition, the color combinations used aim to not distract or exhaust the user while using the system. In addition, we applied design for error mechanisms inside the system, providing interfaces that contain constrains wherever possible. In addition, we attempted to make those interfaces self-explaining, in order not to confuse, but actively assist the user. In order to test the usability of AmITest and improve the UI design, an expert users' evaluation was conducted. During this process a number of UI problems were identified for improvement of the system, which were taken under consideration.

AmITest is, in practice, a core component of the AmI Solertis. It consists of a number of components for testing, being a complete testing toolkit, but also interoperates with other core modules of AmI Solertis for its optimal functionality.
The next sections describe the main components of the AmITest and its collaboration

with other components of AmI Solertis during its workflow process. . We will analyze those components in detail in the next sections.

The testing procedure is split into three main processes: *test composition, execution*, and *reporting* process. The first phase is mainly guided and performed by components related to test composition, such as the Test Composition Wizard. Coming to the execution process, a sandboxed environment needs to be set up, deployed with certain lifetime and produce information-rich results. This environment needs to be orchestrated with precision. After the execution of tests, the reporting process takes place, providing the test execution results to the tester via a rich Graphical User Interface (GUI), but also providing the results to AmI Solertis for further analysis.

The architecture of AmITest is shown in the figure below, and is described in detail in the next sections.

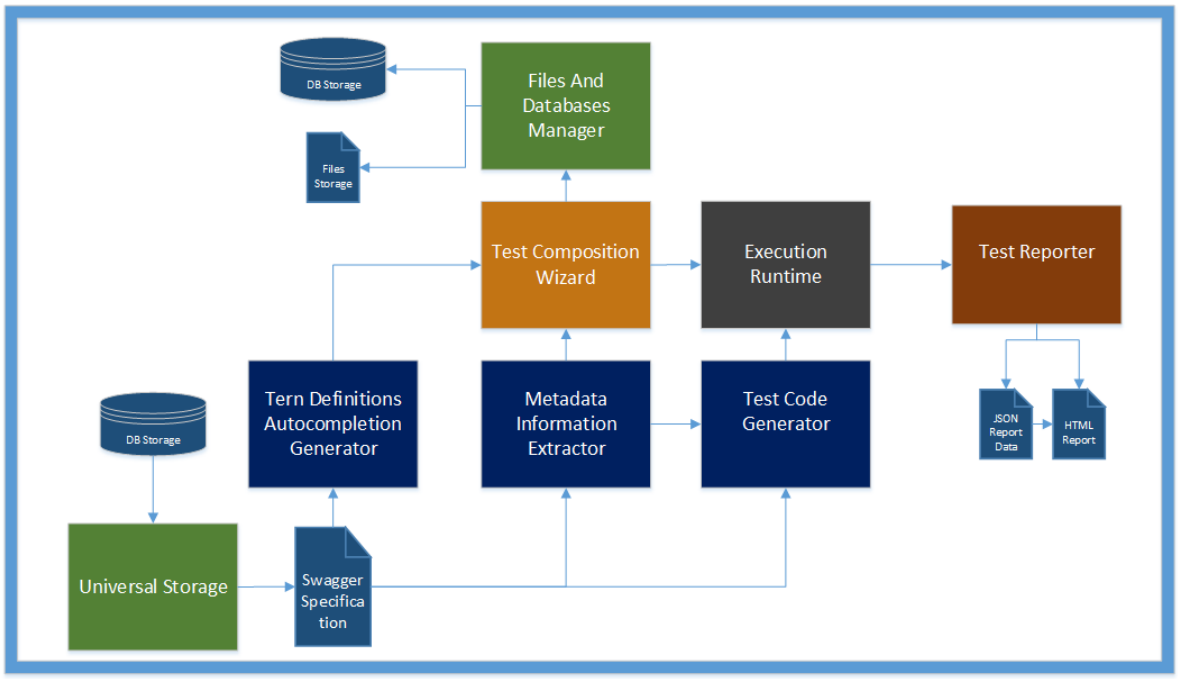

Figure 9 - AmITest Architecture

3.1. Test Composition Wizard

i. Overview

We mentioned earlier that in order to test a real AmI ecosystem, a sandboxed AmI environment needs to be specified. But in order for this to be achieved, this environments needs a considerable amount of information, information that practically need to become specified in detail from a tester during a preparation phase, and then subsequently be executed. This will result in the instantiation of a sandboxed environment that contains a number of simulated or actual user actions, a number of cause and effect operations related with the user actions, and a number of validity checks. Eventually, the system will provide reports for the users and the orchestration system regarding the validity checks in the reporting phase.

The Test Composition Wizard is the main component for the AmI environment tests composition. It consists of a number of front-end components, in order to provide test composition, programming and management capabilities. Its main aim is to guide the potential "tester" of the system in order to compose a test in the most efficient way possible. This is performed via a step-by-step Graphical User Interface (GUI) wizard, which consists of a number of steps that the tester must complete in order to compose the test. In that way, the aim is that the test composition procedure is highly simplified, that a large portion of the test composition complexity is hidden, reducing the amount of information that the tester needs to learn – especially if he/she is an IT "novice", knowing very little from programming and testing. In addition, this approach prevents the developer from making a large number of errors in the test composition. Imagine the scenario that a tester wants to write a typical test script from scratch. He/she has to learn at least the basics of the programming language basics in which the test will be written (in our case, JavaScript) and, in addition, spend some time getting familiarized with the appropriate testing and assertion frameworks. Even if he/she manages to grasp all those information or, even if he/she knows them already to can use them effectively for test programming and composition, he/she will have to grasp the whole image of the ecosystem and a large portion of the internals of its artifacts, but also study at least on the basics of how AmI Solertis works and then attempt to write a test. But even if we make the assumption he/she has grasped this

huge amount of information, he/she can simply do a mistake, in syntactical or logical level.

The aim of AmITest is to hide both the AmI environment and the AmI Solertis orchestration studio internals and their complexity, and provide a solid, straightforward way for the composition of tests. AmITest though does not prevent experienced users from using the internals of the ecosystem and the APIs of AmI Solertis – instead, a large portion of its modules and components provide an API for external usage, in order to make the system fully configurable and extensible for experienced developers who want to perform complex procedures. But for the average developer, the aim is to hide all unnecessary complexity, provide the basic tools for testing composition and assist him/her, by setting this composition to be a step-by-step procedure. In addition, AmITest provides mechanisms for minimizing the possibility of errors, such as a puzzle-like test composition component for end-users with very little or no programming knowledge, and a sophisticated editor component for more experienced users that supports syntax highlighting and hinting, but also extended, dynamic autocompletion capabilities, in order to simplify the process even for experienced developers. AmITest also follows basic design-for-error principles, guiding the user and not allowing him to skip the completion of important information into the test composition process.

As a summary we could say that the wizard ultimately plays the role of the collector of all the appropriate information, in order to specify the sandboxed testing environment, as stated above, yet in a user-friendly way, hiding all the necessary complexity from its end users.

ii.Basic Information Collection

As a first step, the user is guided on a screen that contains an overview of the existing testing scripts. The user is able to modify a script, or create a new one. In case of creating a new one, the user is prompted into the test editing page. Via that page, the user is set to a step-by-step procedure. At first, the developer is requested to complete

the basic test script information, such as the test script name, its description, but also the behavioral scripts that will be under test.

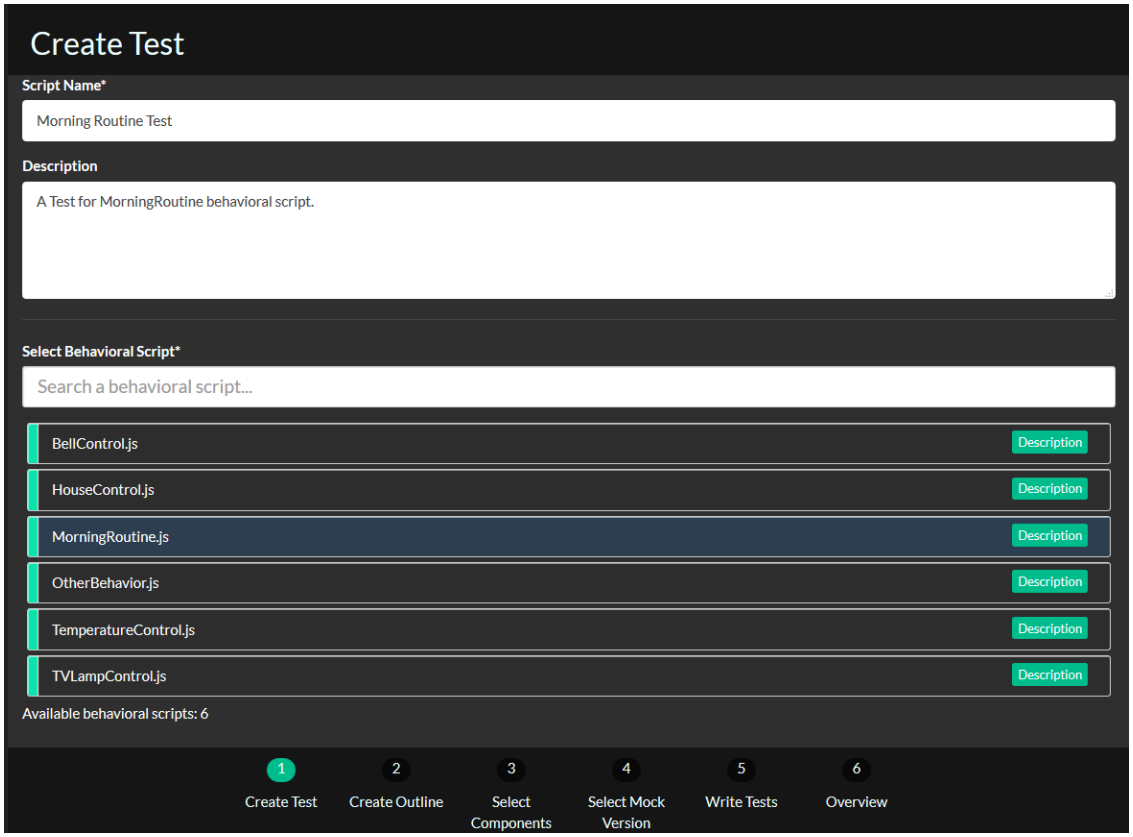

Figure 10 - Test Creation Step

iii. Test Outline Definition

In a second step, the user is requested to specify the outline of the test; in that way, the user can group the tests of the script in different categories that will allow the better management of the tests including their partial activation and deactivation regarding the execution process.

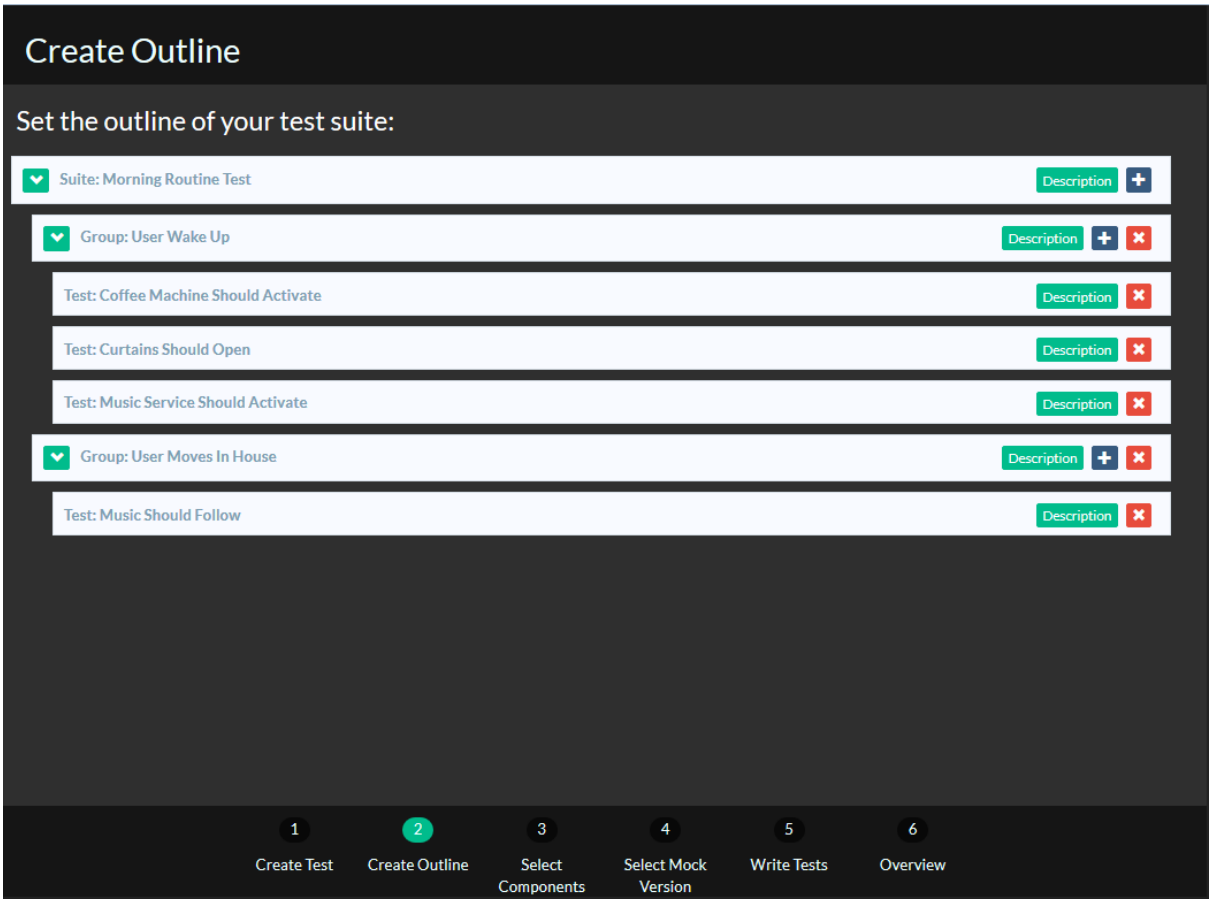

Figure 11 - Create Test Outline Step

iv. Artifacts Selection

In the next step, the user is prompted to select the artifacts of the system that will be used in the test, from a list containing all the artifacts of the AmI ecosystem. It is worth noting that the system itself will attempt to extract information from the selected behavioral scripts via the *Metadata Information Extractor* component after specified in the first step, and, the wizard will provide suggestions to the user about what components he/she can be use in the test, adding them to the list for usage. The user is able to freely add or remove components in that list. In addition, the user is able to specify which components will be considered as *stubs* in the testing process, by just selecting them as stubs.

v. Stub Creation and Selection

After the components specification, the user is prompted into the step of stubs selection. During this step, the user is prompted into a mini-wizard, on which he is

able to specify a stub for each one of the artifacts that he/she selected to be stubs on the previous step. The user is able to select from existing stub versions of a specific artifact, or create new of his/her own. To this purpose, AmITest uses a service outline for each artifact, extracted via the *Metadata Information Extractor* component from the Swagger Specification defined for each artifact in AmI Solertis (the component will be presented below in detail). Using that outline, a (primarily advanced) user can create a new stub service, using an online editor, based on a component called JavaScript Coding Editor based on the CodeMirror library, which provides advanced editing capabilities such as code highlighting, code hints and advanced autocompletion.

The mini-wizard guides the user in order to specify a stub service for each and every artifact specified previously as a stub.

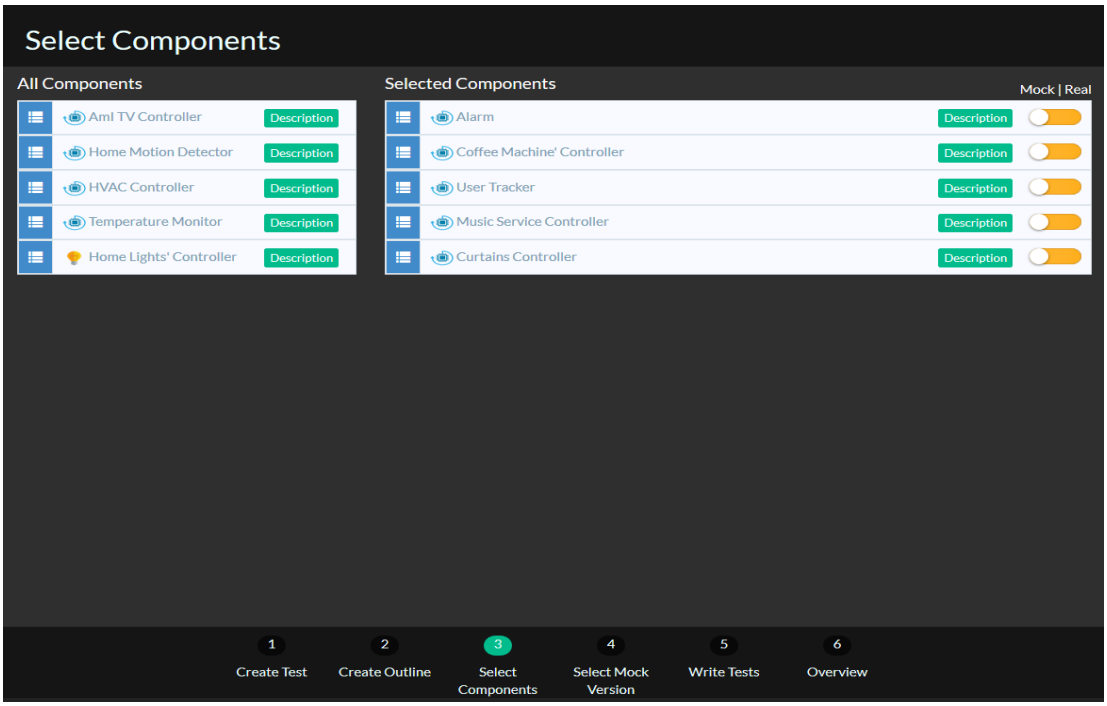

Figure 12 - Artifacts Selection Step

vi. Test Composition

In the next step, the user is prompted to create the tests. This screen can be considered as the "heart" of the wizard. In order for the user to properly construct the tests, following the test structure previously defined, the user is guided by another mini wizard, to each of the test groups specified previously in the test outline. For each one of those steps, the user practically is guided to write the test cases, that have the form

of “cause and effect”: In general, the user is called to specify a number of actions that will cause some effects inside the “sandboxed environment”, and then assert and validate those effects via a number of rule definitions. This procedure can practically be performed in two steps: *block construction* and *coding*.

In the block construction option, the mini-wizard provides a block construction editor, via a component called *Tests Visual Composer*. This composer is based on a modified version of Google’s Blockly framework, in order to provide block composition capabilities. That way, a user with little or no programming experience is able to specify actions for the sake of tests, such as the entrance of a user inside a room, or the enabling of a lamp, and then compose simple “check if *<component.method>* is <status>” blocks. These blocks will eventually be used in order to generate the tests code. In the first version of AmITest, the generated code is based on the Mocha testing framework and the Chai assertion library, while the artifacts are communicated via their proxies defined inside AmI Solertis. It is worth noting though that the system architecture allows the usage of other testing and assertion frameworks and libraries too, as the system is easily configurable, and the code generation procedure is based on a templating mechanism via the *Test Code Generator* component.

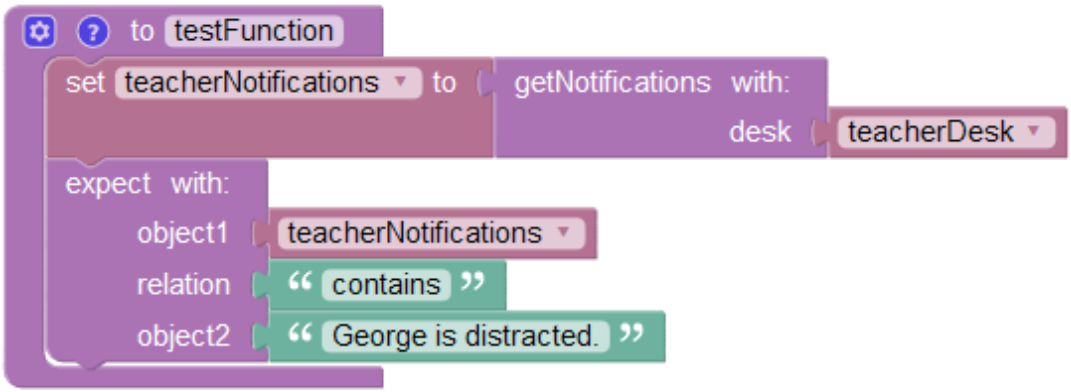

Figure 13 - A Blockly Code Block

The coding option is based on the JavaScript Coding Editor component mentioned above, in order to provide code composition and editing capabilities to users of the system with more advanced programming and testing skills. The user is provided with a basic test outline pre-filled based on the information that has been specified in previous steps, such as the components of the system, which are imported to the script as JavaScript modules, as shown in the figure 14 below.

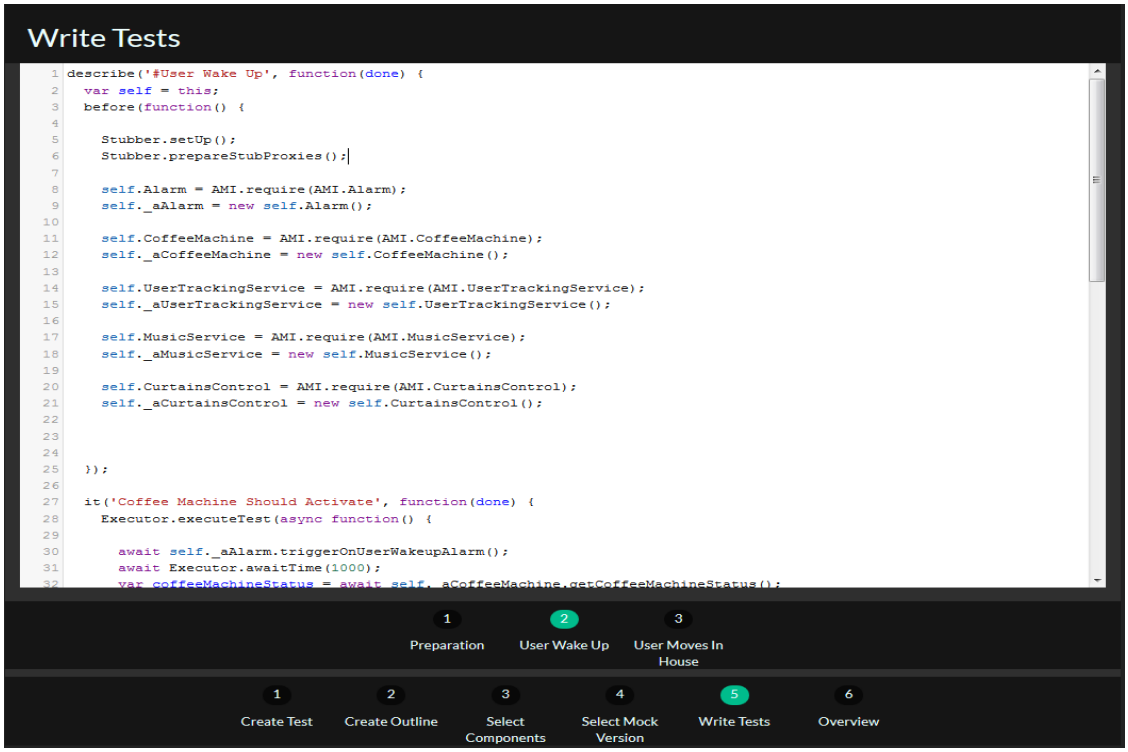

Figure 14 - Test Writing Step - with prefilled code

In addition, Mocha and Chai frameworks but also the artifacts APIs are fully supported into the autocompletion system of the editor, assisting the developer in the testing composition procedure as much as possible. In addition, the developer is provided with a simple API for straightforward asynchronous operations such as time waiting and assertions grouping, called the *Executor API*.

Via the coding option, the tester is able to apply calls to the actual or stubbed system artifacts and apply assertions. In addition, the tester is encouraged to use the async/await JavaScript features, in order to reduce the lines of testing code applied – although the Promise API - a library that provides an implementation for proxies in which a value, not necessarily known when the promise is created allowing the association of handlers with an asynchronous action's eventual success value or failure reason, is also supported by the editor, regarding auto completion and code hints.

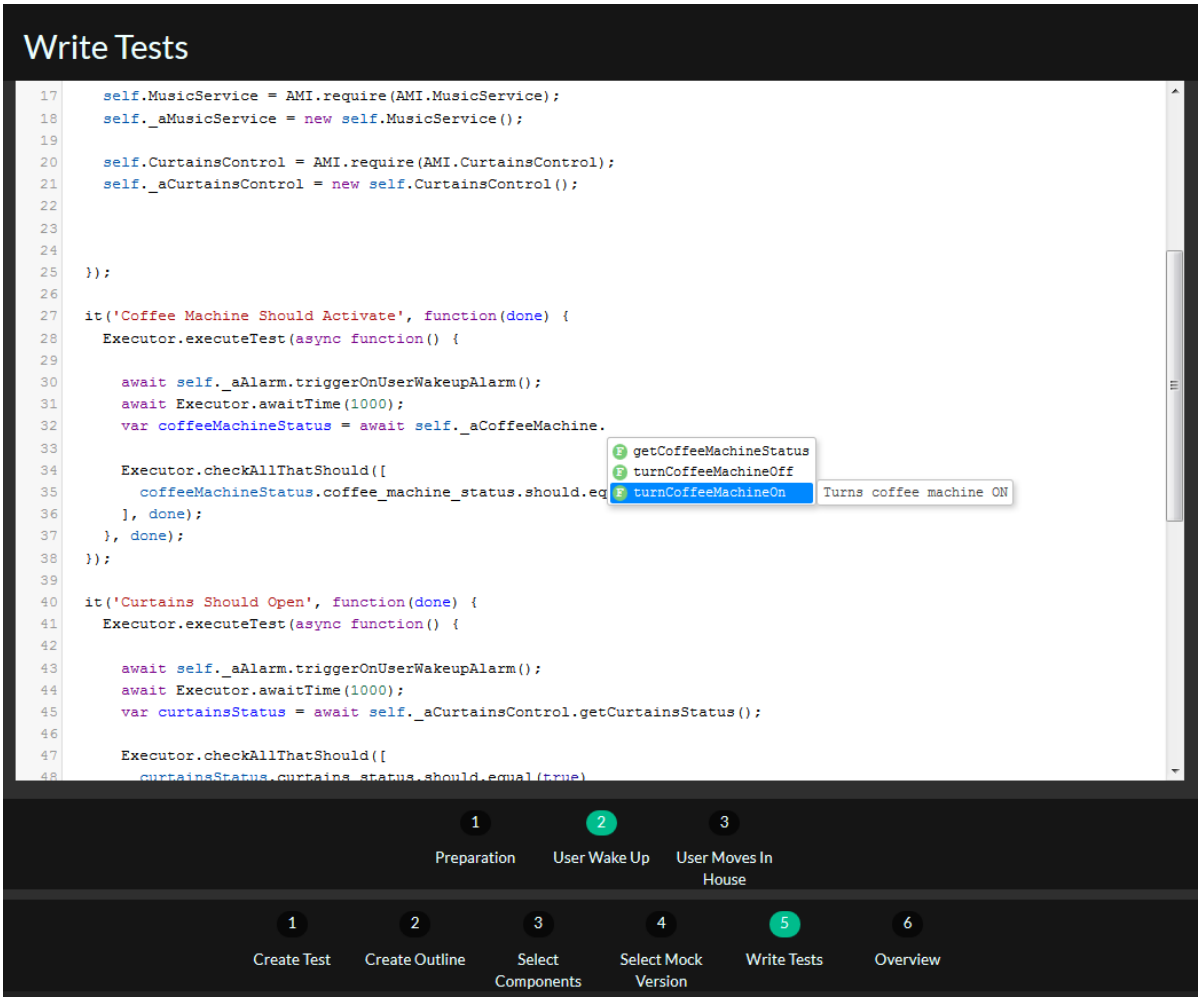

Figure 15 – Test Writing Step - Autocompletion

vii. Test Overview

In this final step, an overview screen is presented to the user, where the user can quickly overview the basic information and structure of the test specified, and apply final modifications to the test, such as the enabling or disabling of test groups for the final execution. As a final step, the user is able to either save the created test to a test scripts repository, or save it but also execute it, leading to the setup of the normal or sandboxed environment, execution of the tests specified as active, and provision of the reports to the end user.

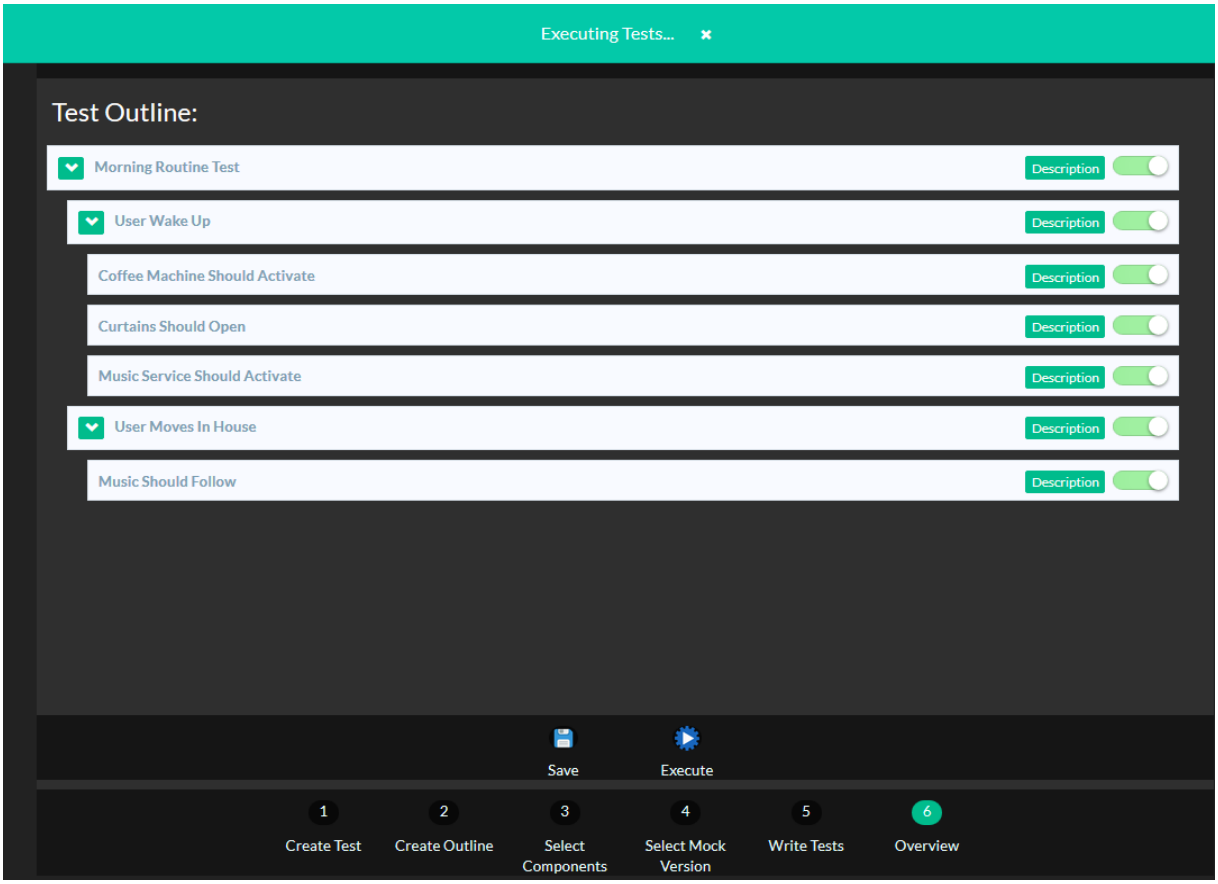

Figure 16 - Test Overview Step

viii. Technologies Used

The Test Composition Wizard is built in AngularJS [52], a very powerful JavaScript – based front-end framework, that genuinely supports Model-View-Controller Architecture [53] for better code composition, containing powerful mechanisms such as two-way data binding between models and views, a very powerful expression mechanism for HTML injection and many other features, setting in a game changer to modern Web Development. The logic behind the wizard is that it practically is a Single Page Application (SPA) that composes an object that contains all the information collected in the specific steps of the test composition process. This object is constructed and enhanced step-by-step, and is then provided to other components that can use it, such as the Files and Database Manager that manages its permanent storage and access when requested or the Test Code Generator for the testing scripts generation in their final module form, using Mocha and Chai.

3.2. Files and Databases Manager

Taking into account that AmI Solertis uses a considerable amount of data, such as AmI ecosystem artifacts metadata, etc., but also generates data such as testing scripts information along with the necessary metadata and dependencies, the system needs to store and access those data on demand. To this end, a component named Files and Databases Manager (FDM) is responsible for the storage capabilities inside the system. In practice, FDM acts as a bridge between a Non-Relation Database Management System, the MongoDB, but also the file system of the server installed, providing a complete API for Create-Read-Write-Delete (CRUD) operation between the databases related on the testing system, but also the filesystem of the server. FDM provides an API so that other components can use it for CRUD operations. In fact, other components rely on the data storage and access capabilities of FDM, such as the Test Composition Wizard (TCW) which uses it in to retrieve artifacts metadata and provide them to the user in the test composition process, but also to store data and information collected from the user, provide editing capabilities, etc. Another component worth mentioning for its extensive usage of FDM is the Test Generator Component (TGC), which requests data on demand from the FDM, in order to generate the appropriate tests and provide them to the Execution Runtime. In addition, Execution Runtime sub-components use the FDM in order to access necessary data, such as the Stubs and Mocks Manager, which accesses the Stub scripts in order to initiate services. Finally, the Reporter use FDM in order to store reporting data for later usage.

3.3. Test Code Generator

AmITest is a framework that attempts to save time, hide complexity from the developer, but also assist him in the testing procedure. Therefore, a number of processes need to become automated, visual programming becomes not only usable but crucial for novice testers, and advanced development editing capabilities need to be supported for more experienced users. All these requirements have a common

base: they all heavily rely on the procedure of *code generation*. In order to automate procedures, automatically generated code could do the job in many cases, such as test snippets that check the availability of an artifact. Such a procedure can become very time consuming for a tester, while it requires a certain level of experience. In addition, complex procedures such as the generation of large API Specification files can also become automated, hiding a large portion of complexity from the testers. In addition, code generation can assist in the provision of code editing capabilities such as autocompletion. To achieve all the above, AmITest contains a core, sophisticated component, named Test Code Generator (TCG). TCG performs a number of tasks, including but not limited to those described above. We will describe its main roles below. TCG consists of a number of sub-components, available for each specific task in the testing environment.

i. Swagger Generation

As we mentioned in the AmI Solertis concept, each artifact can use a number of diverse technologies, yet it follows a standard service API Specification, called Swagger Specification. Swagger definition is considered strict, and can become a time consuming, repetitive and exhausting task for a developer to use it in order to describe an API, especially at larger ones. AmI Solertis aims to automate the Swagger production process. While this is not possible in all cases, it becomes very helpful for a number of them. To this purpose, AmI Solertis supports the analysis and automatic Swagger Definition generation of some artifact types, such as those based on other Middleware technologies, like FAmINE [18] but also other behavioral scripts, exposed as potential artifact services. In the cases that an artifact developer needs to write the Swagger Definition by hand, AmI Solertis provides a sophisticated Swagger Suggestion Editor that supports suggestion and auto completion capabilities, aiming to simplify the process.

The task of the main Swagger Specification is mainly a part of the Solertis API Extractor components.

The AmITest TCG aims at the enhancement of the generated Swagger specification with more test-related information, in direct collaboration with the API Extractor.

ii. Stubs API Generation

A Swagger definition can provide a number of useful information about the testing system. One of them is the precise description of the API outline of each artifact.

Using that outline as a prototype, the AmITest can automatically generate the outline of the Stub definitions, actively assisting the advanced testers in their composition by hand. In addition, as this comes in combination with powerful code editing capabilities, advanced users can be assisted effectively in order to write the proper stubs for each testing process. Similarly to the generation of proxies, TCG uses Handlebars for automatic generation of code.

iii. Tests Code Preparation

Considering that the user follows a step-by-step test composition process, the information provided in one step can be used in later steps for the assistance of the tester. TCG uses the information regarding which artifacts will be used for the test process based on the selection of the user on the Artifacts Definition step of the Tests Composition Wizard, in order to generate preparation code or visual blocks for the developer, in order to assist the test composition process.

iv. Artifact Proxies Generation

In order to provide both programming and testing capabilities in AmI Solertis and AmITest via JavaScript, the actual, physical (or stubbed) artifacts of the system must be mapped in code, like programming entities. In order for this to be achieved, each object is mapped to JavaScript via a JavaScript proxy module, which acts exactly as a proxy adapter between the JavaScript code which uses the module and the service calls to the API of the artifact. These proxies are generated automatically by TCG, via templates.

In fact, TCG uses the HandlebarsJS JavaScript library [54] in order to generate code based on a template. After this operation, the modules are provided to both AmI Solertis and AmITest for further usage. Below, you can see the code of the Handlebars template used for the generation of proxies.

```
/*jshint -W069 */
/**
 *
 * @package {{moduleName}}
 * @param {string} [domain] - The project domain
 */

module.exports = (function () {
  "use strict";

  var amiModule = {};

  // AJAX framework for NodeJS.
  var requestPromise = require("request-promise");

  var Base           = require("./Base_proxy");

  amiModule.{{moduleName}} = class {{moduleName}} extends Base.Proxy{
  {{#swagger}}
    constructor(context, channel, postfix) {
      super(context, channel);
      this.postfix = postfix != null ? postfix : "{{moduleName}}";
    }

    {{#methods}}

    /**
     * @method
     * @name {{methodFullName}}
    {{#each parameters}}
     * @param {{name}} - {{description}}
    {{/each}}
     *
     */
    {{methodFullName}}({{#each parameters}}{{name}}{{^isLast}}, {{/isLast}}{{/each}}) {
      {{#each parameters}}
      {{#required}}
      if({{name}} === undefined) {
        return Promise.reject(new Error("Missing required  parameter: {{name}}"));
      }
      {{/required}}
      {{/each}}

      {{#if isEventHandler}}

      // Register callback to Local Events Registry (LER)
      // in order to be called on event dispatching and
      // return a promise on the operation.
      this.amiEventClient.SubscribeTo(this.domain + "_{{methodFullName}}", {{parameters.[0].name}});

      {{else}}
      {{^dataInBody}}
      var headers = {};
      {{#each parameters}}
      {{#isHeaderParameter}}
      headers["{{name}}"] = {{name}};
      {{/isHeaderParameter}}
      {{/each}}
```

Figure 17 - HandlebarsJS Template for Proxies Generation (partial)

v. Test Composition

Another very important task of the TCG, is the test composition. As we mentioned in the previous section, Test Composition Wizard (TCW) is practically, a User Interface component that aims to collect all the appropriate information from a user's perspective in order to compose a test. Furthermore, the test composition process has to be performed and eventually result in a test that will be executed by the Execution Runtime of AmITest. In practice, this means that one has to do the job of getting all the information collected from the Test Composition Wizard, combine them with some meta information regarding the artifacts of the system or the system itself, and generate the tests in a form that will be able to be executed by the Execution Runtime component. This task is performed by a component named *Test Composer(TC).* In practice, the TC uses all the data regarding a test script collected from the Test Composition Wizard, including the information regarding the test itself, the artifacts

used inside the test, the referred behavioral scripts, its structure, and of course, its test cases. The TCG also uses data regarding each artifact, such as the JavaScript module paths used in the script, the module names of the artifacts used, etc. Eventually, the Code Generator generates a test script in pure JavaScript code, ready for execution from the Execution Runtime of AmITest. This JavaScript module script uses a well-known framework for unit and integration testing named Mocha. Mocha is a very powerful framework for unit and integration testing, providing great categorization between the tests, and supporting a variety of assertion libraries, such as Chai and ShouldJS. In addition, Mocha supports testing for asynchronous programming. This is a key factor to the selection of this framework, as asynchronous programming is the basis of the orchestration of an Ambient Intelligence ecosystem, as all its artifacts work asynchronously inside it, executing actions atomically, but also observing for events inside the ecosystem that might affect them. Of course, their behavior is based on the APIs they expose, based on a micro-Service Oriented Architecture (micro-SOA) [16], and is orchestrated via the AmI Solertis behavioral scripts. Taking this into account, one can easily infer that the nature of the behavioral scripts, and their referred artifacts is asynchronous. The TCG uses all the information mentioned above to construct test scripts that are able to validate the behavior of asynchronous code and events, written inside one or more behavioral scripts. In addition, it adds a number of automatically generated scripts that check the availability of non-stubbing artifacts, in order to ensure the proper tests execution. Those tests are composed and provided to the system as simple JavaScript scripts (modules), using Mocha and Chai frameworks.

TCG follows here the same policy of the generation of proxies and stub service outlines of artifacts, namely, it uses HandlebarsJS templates in order to generate the code desired.

```
var chai      = require('chai');
chai.use(require('chai-as-promised'));
var should    = chai.should();
var Stubber   = require('../../components/mocks/stubber.js');
var AMI   = require('../../components/runtime/AMI.js');
var Executor = require('../../components/runtime/executor.js');


{{#prepareTestScript}}
{{{code}}}
{{/prepareTestScript}}

setUp();

{{#testStructure}}

{{#each nodes}}

{{#if isEnabled}}
{{{code}}}
{{/if}}

{{/each}}

{{/testStructure}}

tearDown();
```

Figure 18 - HandlebarsJS Template for Test Generation

At this point, a serious consideration must be made: in order for the test to be performed in an effective way, the Ecosystem State Modification Commands (ESMCs) and the Test Assertion Commands (TACs) must be performed in a sequential way, despite their asynchronous nature. While this is an oxymoron, such sequence must be observed, in order to attempt to avoid race condition between artifacts, but also the execution of assertion code without waiting for an artifact under test to reach to its desired state, due to facts related to parallel programming – such as network delays, etc. For instance, a test could attempt to activate a lamp and then validate that its status has been altered to active, using an assertion. Considering that the lamp provides a REST API accessible via network, the call that activates the lamp is asynchronous, and therefore there is a possibility that when the assertion is executed, the lamp status has not been altered on time. Therefore, a mechanism must be applied, in order to ensure that the asynchronous nature of such calls is taken into account. Currently, there are two ways to achieve this in the JavaScript language: via the Promises JavaScript API, or the async/await keyword feature. A Promise [21] is a proxy for a value not necessarily known when the promise is created. It allows the

association of handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods. Instead of immediately returning the final value, the asynchronous method returns *a* Promise object in order to supply the value at some point in the future.
The async [22] function declaration defines an asynchronous function, which returns an object that represents the code executed within such asynchronous function (a Promise object). This is commonly used in combination with the await operator [23], which is used in order to wait for a Promise object returned by an async function.
The first way is much more stable, tested and supported, yet it leads to complex Promise chaining, leading to non-flexible, less readable code and generating much more lines of code. The second way is much more robust, generating less lines of code, but is still in experimental usage in JavaScript, aiming to become a standard (Stage 4 on release at the time of writing).
Ideally, the execution of ESMCs should *always* come after the execution of TACs, so that race conditions are avoided in the best way possible.

Of course, such a statement is much more complex to be proved than simply mentioned. Considering that the system states can be abstractly presented in a state graph, an ESMC can lead to the generation of many, diverse, phenomenally unrelated events in an asynchronous manner, then this could lead to race conditions inside the system and produce many states, including state cycles regarding the state of its artifacts. Considering though that the TACs followed by ESMCs aim to validate specific artifact states inside the system, the system is interested to ensure that those states in particular have come to a stable unchanging state, regardless of the whole ecosystem. We also assume that cyclic states generated asynchronously as a result of an event can indicate problematic behavior. We will further discuss this particular issue in the challenges section later on.

vi. Visual Language to Code composition

Another worth noting task of the TGC, is the analysis of data collected from the Visual Editor Component of the TCW. The Visual Editor practically provides a block construction user interface for the test composition. This block composition UI hides the asynchronous code connection complexity – the blocks are connected as normal,

serial instructions. The Test Code Generator analyzes the blocks constructed in a sequential way and generates the asynchronous code. As mentioned above, this is performed by using the async/await JavaScript feature, mentioned above.

3.4. Tern Definitions Autocompletion Generator

We mentioned the case of test composition via a Visual Programming tool, and the crucial role that the TCG plays in the proper test code composition. We must consider though that Visual Programming mainly addresses novice users, and that we expect more experienced users to attempt to write their tests via coding. In that case, the tester is able to use the Code Editor in the TCW, and it is his/her responsibility to perform a number of complex operations, such as adding the sequential code behavior described above, using async/await or the Promises API, both supported by AmITest. Considering that this is not an easy task, AmITest provides a number of code editing capabilities, via its WYSIWYG editor component, used across the AmITest framework, supporting important editing capabilities. One of those capabilities, is advanced code auto completion, and is provided by using a powerful auto completion engine, named Tern [55]. Tern uses a number of definitions, based on JavaScript Notation (JSON) [56] format and an expression language in order to provide the mentioned auto completion capabilities. Those definitions are automatically generated by TCG when a component is registered to AmI Solertis.

The TCG is based on templating in order to generate the appropriate script code. To this purpose, the HandlebarsJS framework is used, a framework that allows powerful templating capabilities via special expressions, similar to those used in AngularJS [52]. In contrast with many code generation tools, the TCG attempts to generate as human readable code as possible, considering that the tests it generates should be suitable for editing. Considering that AmITest should give as much control to the tester as possible, we plan to add editing capabilities to the finally generated tests, so that an advanced tester should be able to apply advanced modifications to the script before its execution.

Figure 19 - Tern Definition of a Light Control Proxy

3.5. Metadata Information Extractor

An Ambient Intelligence environment consists of a number of artifacts that contain a large variety of properties. In order for AmITest to be able to put them under test, a considerable amount of artifact-related data must be collected and used. In addition, tests contain information regarding them, useful for the testing process. All these information needs to be extracted from the appropriate resources in order to be used in the testing procedure. AmITest contains a mechanism for artifacts metadata extraction, a powerful component, called Metadata Information Extractor (MIC). This component works in collaboration with other components of AmI Solertis, such as the API extractor. MIC extracts all the metadata provided by an artifact, such as those provided through their Swagger API specification, their API JSDoc [57] definitions etc. After an extensive analysis is done on artifact specifications, metadata are fed into other systems, such as the Tests Code Generator, aiming to provide better code generation results.

3.6. Execution Runtime

After a test is composed and all the appropriate code regarding the stubbing of its artifacts and the tests is generated, the test must be executed. While it seems simple, such a procedure is not straightforward. In fact, a "brain" entity needs to consider all the information collected via user interaction and the AmI ecosystem artifacts metadata, use this knowledge in order to prepare and initialize services required for testing, ensure the proper communication of AmI artifacts under test with the testing environment itself, deploy the behavioral scripts under test inside a sandbox, and then execute properly all the test preparation behaviors and the testing assertions, in order to produce system validation results, that will be fed to the components which manage and work on the proper reporting components, informing the user, the ecosystem, and the whole orchestration system. This is part of the reporting process, executed almost in parallel with the execution process. The "brain", is actually a component named Execution Runtime (ER), which in practice sets up the sandboxed environment by preparing, performing and managing the tasks described above.
All the tasks of the ER are performed by a number of sub-components, each one assigned a specific task. Those components are described in detail below.

i. Stubs and Mocks Manager

One of the most crucial tasks the ER performs is the preparation and execution of Stubs and Mocks. As we mentioned earlier, while testing, there are cases that the behavior of some artifacts of the AmI ecosystem need to become mocked in a very specific way. The Stubs and Mocks Manager (SMM) is a component responsible for managing a number of stub services that will impersonate the desired artifacts within the ecosystem. The SMM is responsible for the proper setup of each stub service, configuring settings such as its starting process information, its endpoint information configuration (service port and more), etc. As a follow up, the SMM is responsible for the proper initialization of the services, handling their lifetime and ensuring their proper functioning throughout the whole testing process, but also handling erroneous situations, propagating the proper error messages to the reporting interfaces.

The current prototype of AmITest mainly supports Stubs, aiming at the impersonation of the API of existing artifacts. In future version, testing capabilities on the impersonators itself will be added, such as capabilities for the number of calls of an API function, etc., consisting of components called Mocks. This component will be responsible for their management and execution, as their behavior is very close to mocks.

ii. Behavioral Scripts Executor

In order for the behavior of a script to become evaluated via dynamic analysis, it must become executed. Considering though that a faulty and problematic script could cause inconsistencies and problematic behaviors inside the ecosystem, it may be necessary for the script to be tested in an isolated, sandboxed environment, and that its behavior should not affect the behavior of the real system, unless this is performed purposefully. The Behavioral Scripts Executor (BSS) is a component responsible for deploying, executing and monitoring the testing scripts in outside the actual ecosystem, in a sandboxed environment. The BSS is also responsible for its proper execution and automatic monitoring. In practice, a behavioral script is deployed and executed in after the initialization of the stub services. This leads to the restriction of the behavior of the script, allowing its behavioral validation without affecting real artifacts of the ecosystem, unless desired; if the tester does not specify a component as a mock, then the behavior of the script will affect the real artifact. This allows the tester to apply validity checks to specific real artifacts of the ecosystem, in order to ensure their proper behaviors. The tester can also test the system's behavior under a specific situation, where an artifact is not activated in the system, yet it is deployed. In this case, the tester can create a mock version of the deployed artifact and check how the system performs.

iii. Testing Commands and Assertions Executor

After the SMM and BSS have prepared and initialized the stubs and the behavioral scripts under test, the actual validation procedure needs to take place. The execution of Ecosystem State Modification Commands (ESMCs) aim to lead to implicit or explicit modifications of the state of some or all the artifacts of the ecosystem, and, therefore, modify the ecosystem itself. For instance, a command that simulates a user who has entered a room of an AmI House is an ESMC, as it will potentially lead to a series of events inside the ecosystem, and therefore, modify its state. After the execution of ESMCs follows the execution of Test Assertion Commands (TACs) that validate that some artifacts inside the ecosystem have come to a specific state, practically validating that the command executed leads the ecosystem to a valid state, depending on the behavior defined in the behavioral scripts under test. Notice here that the term "artifact" can indicate either a real ecosystem artifact, or its stub, that simulates its behavior. Such distinction is dependent on the user options definitions in the test, as specified in the Test Composition Wizard. In practice, a component named Testing Commands and Assertions Executor (TCAE) receives a test script, composed by the Test Code Generator (TCG) component, using data from the Test Composition Wizard (TCW) and the Test Metadata Extractor (TME) components, which will be analyzed in detail below. The TCAE then initializes a NodeJS process and executes the composed test, which contains both ESMCs but also the related TACs. In addition, the TCAE notifies the components related with the reporting process for any potential outcomes of the tests, based on the execution of ESMCs and TACs, information which they use in order to provide reporting features to the system testers, the ecosystem users, but also the Solertis Orchestration System itself for further analysis.

A summary of the whole work of the Execution runtime can be seen in the command prompt shown in the figure 20 below.

```
ALIAS: CoffeeMachine
ALIAS: UserTrackingService
ALIAS: MusicService
ALIAS: CurtainsControl
> GET:  http://127.0.0.1:62218/amitest/curtains/open
> GET:  http://127.0.0.1:62218/amitest/curtains/status
> GET:  http://127.0.0.1:62217/amitest/music/turnON
> GET:  http://127.0.0.1:62217/amitest/music/toBedroom
> GET:  http://127.0.0.1:62217/amitest/music/status
> GET:  http://127.0.0.1:62215/amitest/coffeemachinecontrol/turnON
> GET:  http://127.0.0.1:62215/amitest/coffeemachinecontrol/status
==> MORNING ROUTINE INITIATED
Curtains: true
Music: true @ bedroom
Coffee Machine: true
> GET:  http://127.0.0.1:62217/amitest/music/status
User is in the living room. Music should play @ LIVING_ROOM
> GET:  http://127.0.0.1:62217/amitest/music/toLivingRoom
> GET:  http://127.0.0.1:62217/amitest/music/status
Music Playing? true
[Output device: living_room]
> GET:  http://127.0.0.1:62217/amitest/music/status
    √ Music Should Follow (2012ms)

/////TEST HAS NOW ENDED!!
```

Figure 20 - Command Prompt from the Execution Runtime Process

3.7. Test Reporter

The ultimate aim of testing is to find potential or already existing flaws on the behavior of the system, and take the appropriate actions before an error leads into catastrophic results.

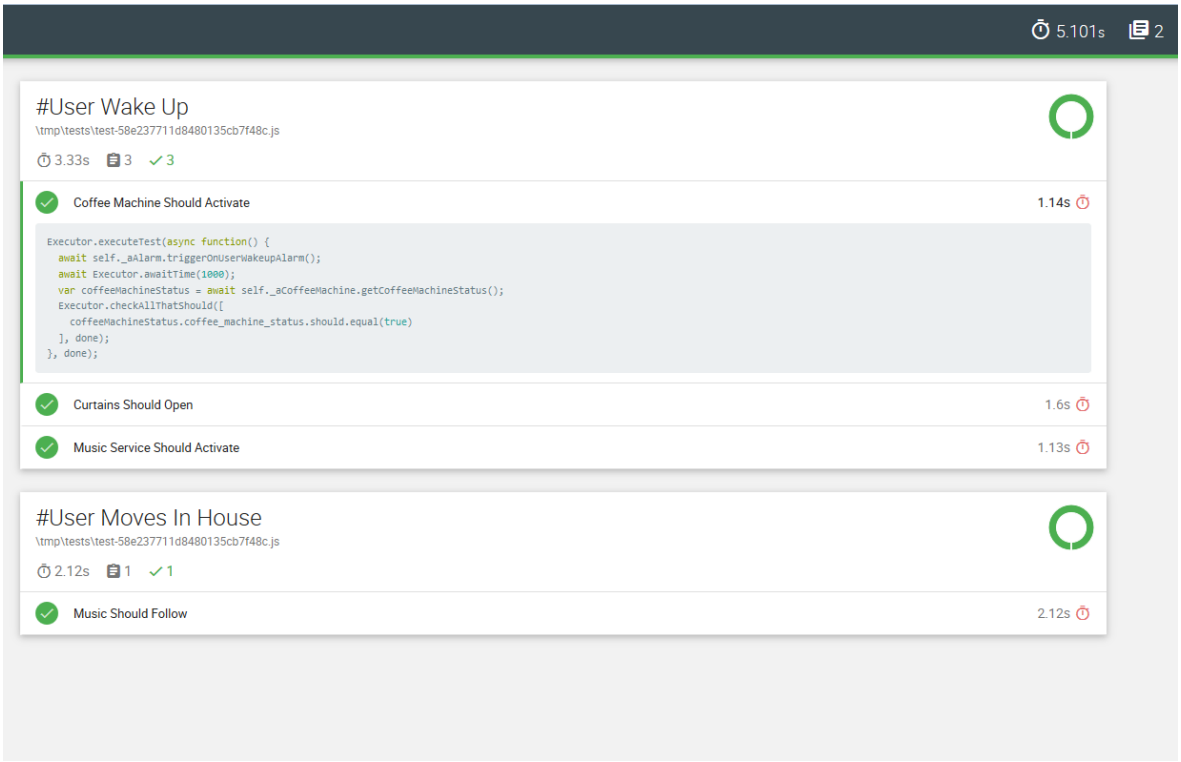

Figure 21 - Test Report – All Tests Passed

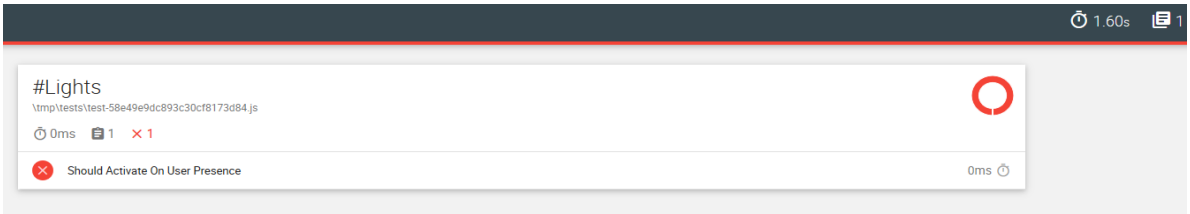

Figure 22 – Test Report -A Test Failed

The purpose of testing is eventually to come to results that will lead to error prevention or error repairing and recovery actions. Those actions could come from users, such as the developers of the system, but also ecosystem inhabitants. Another important purpose is the analysis of the tests results by AmI Solertis. Orchestrating does not only mean to be able to specify strict instructions, but also fix erroneous situations and repair them effectively. The system itself could use test reports in order to proceed to error prevention or repairing actions.

The Test Reporter (TC) is a component responsible for the management and orchestration of the reporting process.

In its most simple task, the reporter provides a set of reporting results through a Graphical User Interface to the testers that execute the test. Those results can contain crucial information about the status of the system, and potential problematic behaviors that one or more behavioral scripts could cause upon their actual deployment inside the real ecosystem.

At a more sophisticated level, RC could provide a number of data to the AmI Solertis ecosystem, in order to assist into the inference of summations of the health status of the ecosystem. This task can be performed in combination with log data and machine learning methods and is considered as a very useful mechanism, planned for future work. The reporter also works in collaboration with the Files and Databases Component in order to store all reporting data for use by analysis components and testers.

The test reporter uses the Mocha reporting messages defined in the Test Composition Wizard, in order to provide the testers with useful information about the test outcome. The Test Reporter works in parallel with the Execution Runtime, providing real-time reporting, and allowing the immediate interaction with the tester.

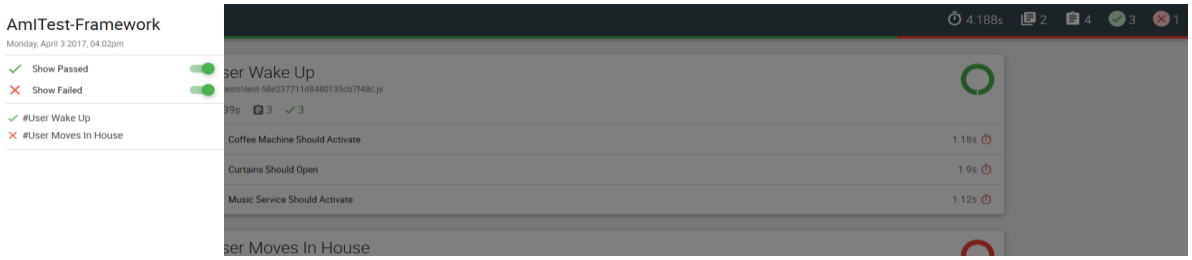

Figure 23 - Among Others, Reporting Allows Filtering Capabilities

3.8. Challenges

One of the most challenging issues in order to perform testing operations is the complexity of the system; there is a considerable number of distributed, interoperating components, applying operations asynchronously between their operations most of the time, but also acting asynchronously between each other. Considering artifacts as isolated units and testing them that way would be incorrect, as there is a high level of dependency between the artifacts.

What we attempted though was focusing mainly on isolating and performing assertions on the values of the artifacts, thus practically checking all the individual components operate as expected. For instance, considering that our test case is inside a Smart Learning Environment, we should consider that if a student gets distracted during a lecture, then the teacher should be offered the opportunity to motivate her to participate. In order to validate that these operations function as intended, one could observe the situation of the class, something that it is not possible in a simulation scenario. On the other hand, this observation could be done via value checking of all the affected artifacts. Therefore a complete test would assert that: 1) the status of the teacher's workstation would change from "classroom overview", to "inattention detected" and eventually to "mini-game launched" and 2) the AmIDesk of the distracted student would disable interaction with every application but the mini-game initiated by the teacher.

In order to achieve the application of assertions in systems collaborating asynchronously, while attempting to reduce the complexity of the code required for validation, state-of-the-art, even newly added mechanisms of JavaScript language had

to be considered and used, such as the async/await mechanism, introduced as a standard on the ECMAScript2017 Draft (ECMA-262).

In addition, we have implemented a sophisticated tests runtime orchestration mechanism through which we ensure that value checking (i.e., Expectations) is performed only after the necessary handling actions have completed (i.e., Promises).

Another challenging aspect of the development of such a testing system, was the provision of a simple, yet effective mocking mechanism of components, in order to provide effective impersonation of the artifacts of the AmI environment under test. While the consideration of the usage of JavaScript modules in combination with mocking libraries such as SinonJS [58] seemed to be an effective way for stubbing, the usage of stubs as services proved to be extremely robust, giving the ability to the tester to practically create a real impersonator of the system with its own service lifetime, runtime properties and full API provision. Of course, such a choice had its tradeoffs, as the mocking procedure increased the execution time of the testing, since practically stubs are provided as service REST APIs and are accessed via network. In addition, such a stubbing mechanism requires the creation of stubs from developers with basic experience in Service Oriented Architecture and REST APIs development. Considering though that the cost of fast internet connections is affordable, the usage of such stubs is limited to specific testing cases, and eventually that stubbing services can be written from experienced developers but used by novice users without problems, to this solution was preferred.

Of course, as our system scales up, even more challenges arise. Inside an Ambient Intelligence environment, human inhabitants get monitored continuously. This leads into discussions of privacy issues, a well-known problem in the field of Ambient Intelligence. While the focus of this work is mainly on the benefits of the testing procedure, it is worth mentioning that our plans include the compliance of our system to all the standards applied in Ambient Intelligence, related to security and privacy. While our system does not focus on the data, we plan to use technologies such as SSL [59] and OAuth 2.0[60] authentication in the usage of our APIs, in order to protect the main AmI Environment from malicious attacks, regarding the areas related to our testing system.

4. EVALUATION

For the purpose of this work, we conducted a user evaluation experiment in order to evaluate the usability of our system, but also collect information regarding the general opinion of the users regarding the system and its use in its actual environment. For this first evaluation, we focused on the use of the system by expert IT users, in particular concerning the part of the system that allows the test composition by IT experts, before moving on to an evaluation with novice users.

To this purpose, we conducted an experiment with 5 users, a number suggested for user evaluation by Nielsen [61]. The experiment included 4 men and 1 woman (a factor considered irrelevant in our experiment), users of ages 25-40 years old. All the participants were developers, having worked in projects regarding Ambient Intelligence, and therefore familiar with the related concept and principles.

We performed the experiment in two phases. In the first phase, we trained the developers in the usage of the system, first by showing them how to write a simple test case for a script that enabled the lights of a room, when the first user entered it. The users were instructed to write a test case that simulated the behavior of the entrance of a user inside the room, and then applying assertions on the status of the lamp. After this example, the users were requested to use the system in order to get familiarized with it, by writing similar scenarios – regarding the deactivation of lights when the last user of a room exits and the activation of the coffee machine when the alarm triggers, assuming the user sleeps inside the room. In order to assist the developers, a single page cheat sheet was given to them. Before this training phase, it was requested from them to answer some preliminary questions, regarding their expertise, but also their approach to validating the behavior of a smart room they had hypothetically programmed. Users were encouraged to make comments even on the training phase. Regarding this question, it is worth mentioning that two of the users mentioned the physical interaction with the actual ecosystem, followed by the reasoning of the correctness via the observation of the state of the actual system.
In addition, all users proposed the creation of a system that would support the feeding of the ecosystem with fake data or the simulation of events inside the ecosystem, and then applying assertions on the eventual state of the system.

The second phase of the evaluation experiment was performed the next day after the training phase for all users, in order to allow them enough time to further grasp the information learned. In this phase, the users were requested to perform two scenarios, one of easy difficulty regarding the testing of the automatic enabling and disabling of air-conditioning regarding the temperature of the environment, and one of intermediate difficulty, regarding the activation when the user wakes up in the morning, along with the validation of the behavior of devices when the user moves from his/her bedroom to the living room. Finally, the experiment included one debriefing scenario for the detection of faulty behavior inside the ecosystem – the case was a (simulated) problematic lamp. The users were encouraged to express their opinions openly, but also to express their way of thinking during the performance of tests (following the think-aloud protocol [62]). Any quantitative data – such as impressions, comments and suggestions during the scenario execution phase were documented, along with a number of qualitative data. More specifically, for each scenario we measured the task completion time required by each user in order to complete it, along with the number of help requests and errors they made in order to complete a task, in three basic categories: Test Procedure/Beyond User Interface, User Interface, and Code Composition. In general, all users managed to complete all scenarios and receive the appropriate results.

We present the diagrams of each case below. The orange color represents the average value, while the green one represents the median, in all diagrams.

4.1. Scenario 1

As we mentioned earlier, scenario 1 was the introductory, warm-up scenario of the evaluation. We observed that users had received well the training of the previous day. The results of its analysis are presented below:

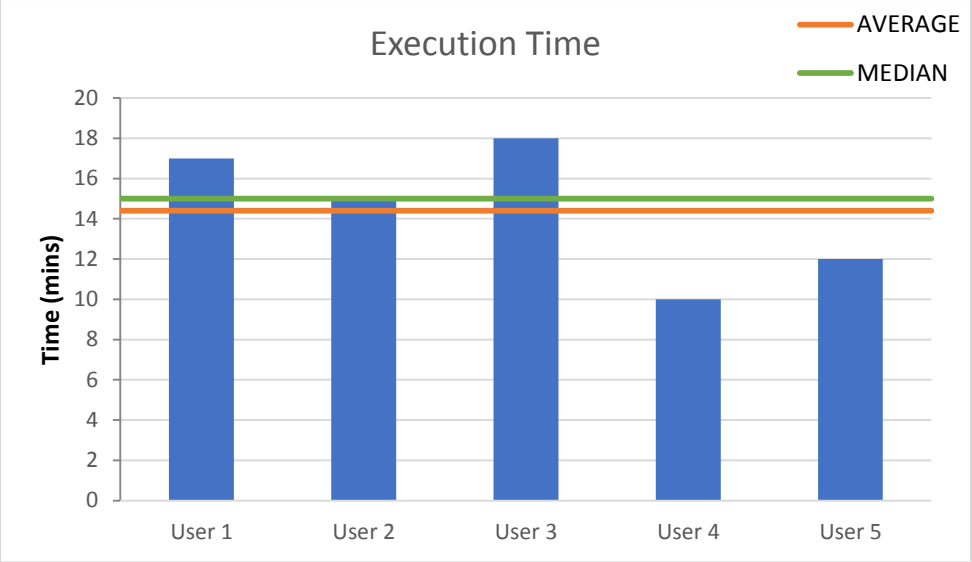

Figure 24 - Execution Time - Scenario 1

As figure 23 shows, all users managed to complete the first, easy scenario air condition functionality without large deviations, except user 4, which managed to complete it in 10 minutes. Considering that such a scenario was written in such an amount of time, the results are promising regarding the usage of the tool, considering that the users were using it for the first time apart from the training session, such a time span should be considered small and expected to reduce with the extensive usage of the system.

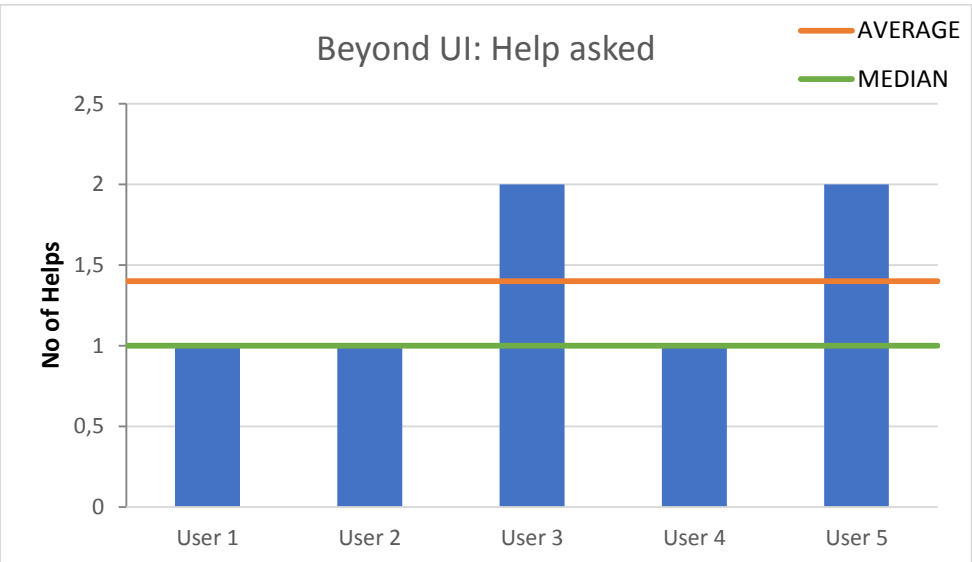

Figure 25 - Help Requested - Process/Beyond UI - Scenario 1

Figure 24 shows that the participants did not request much help regarding this scenario. Based on these results, we can conclude that users understood easily the overall testing procedure and proceeded without big problems and that although everyone asked for help, the amount of assistance remained low. Considering though that all users asked for help at least once, this indicated that our system needs minor improvements regarding the process of test composition. It is worth mentioning though that there are no errors regarding the process, which leads us to assuming that the participants grasped successfully the whole process.

Regarding the UI, the amount of help requested and errors made remains low for all participants (zero to one), which means that the User Interface did not cause any problems to the users for a simple scenario. The same applies regarding to the help requested considering the code composition procedure. What is interesting about code composition is the diagram of errors in code.

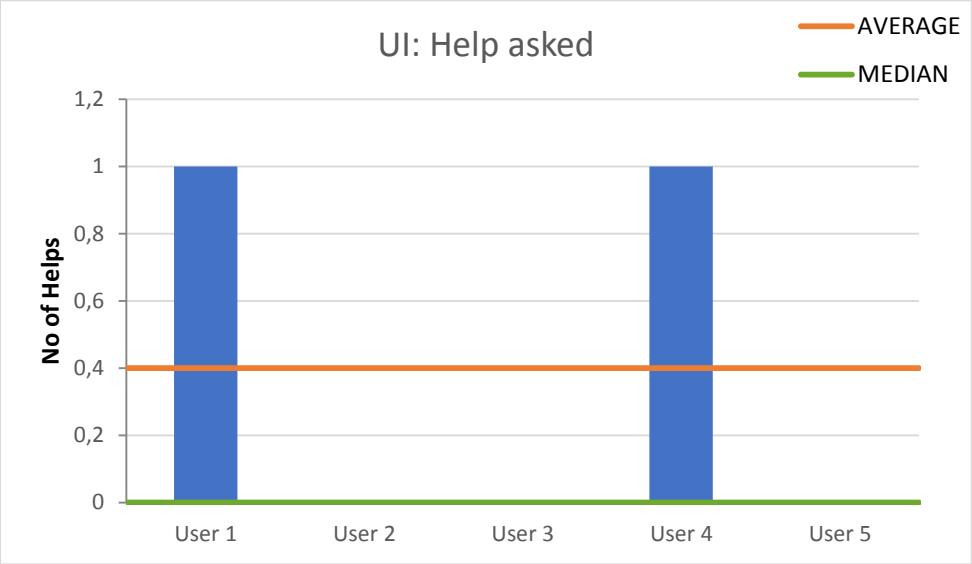

Figure 26 - UI Help Asked - Scenario 1

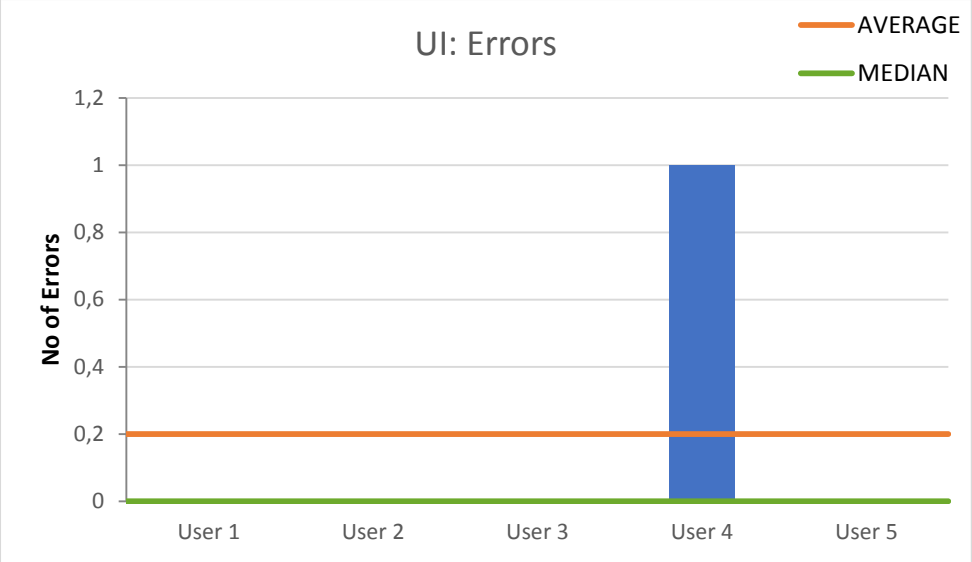

Figure 27 - UI Errors - Scenario 1

Regarding the UI, we can see in figures 25 and 26 that both errors and requests for help are limited for this simple scenario, mainly related to the wrong selection of components because their presentation was not indicating their role clearly, a concern that users also expressed.

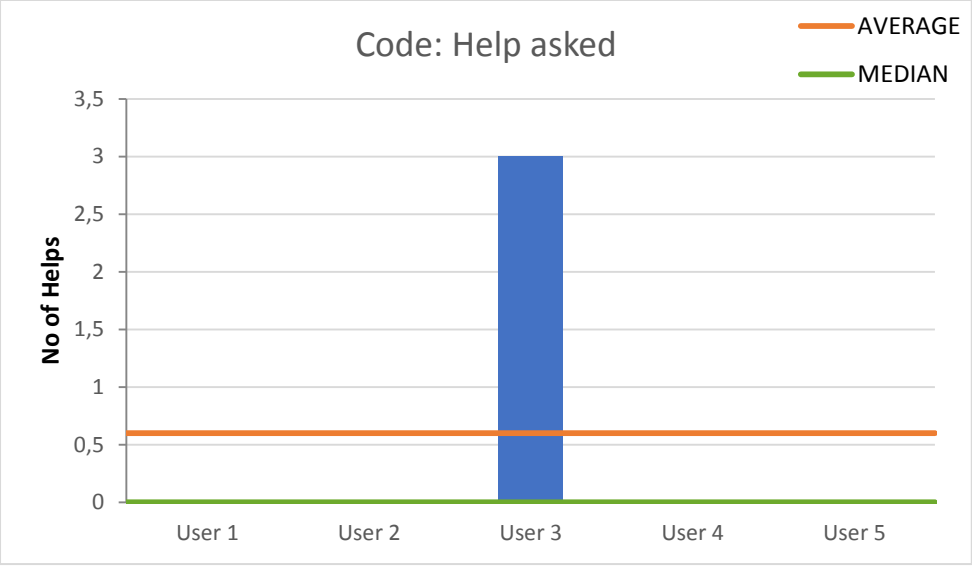

Figure 28 - Code: Help Asked - Scenario 1

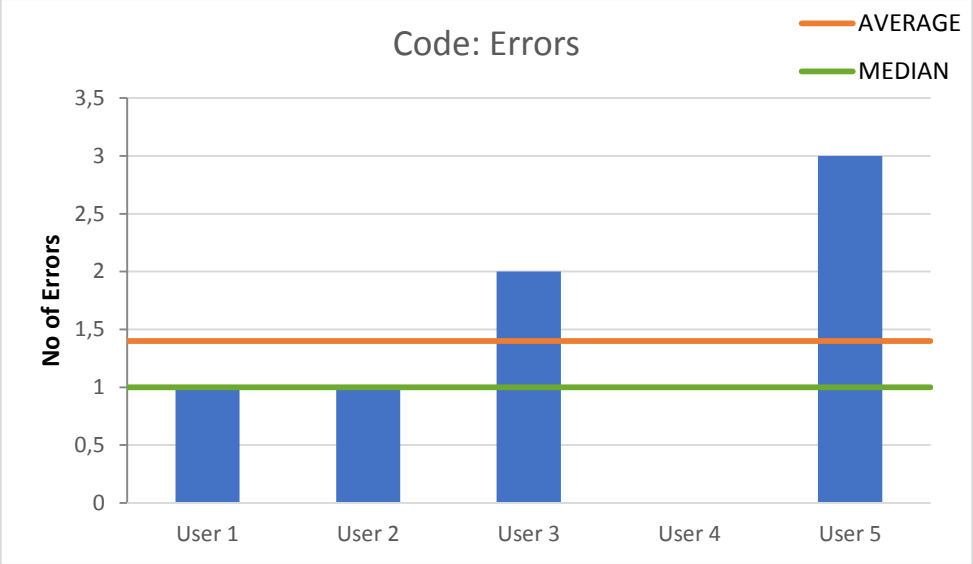

Figure 29 - Code Errors - Scenario 1

Considering figures 27 and 28, we observe that, while only one user requested help, almost all users made some programming errors, despite the simplicity of the scenario and the provided tools. This indicates that the coding process needs refinements.

4.2. Scenario 2

After scenario 1, the participants were prompted to proceed to scenario 2, which was more complicated than the first one. This scenario included the usage of an increased number of artifacts, but also a more articulated scenario which included the activation of a number of devices upon alarm activation, but also the proper detection of the user while moving from his/her bedroom to the living room, followed by the proper activation of the music service in the room he/she has entered. For this scenario we also counted the qualitative measures used in scenario 1.

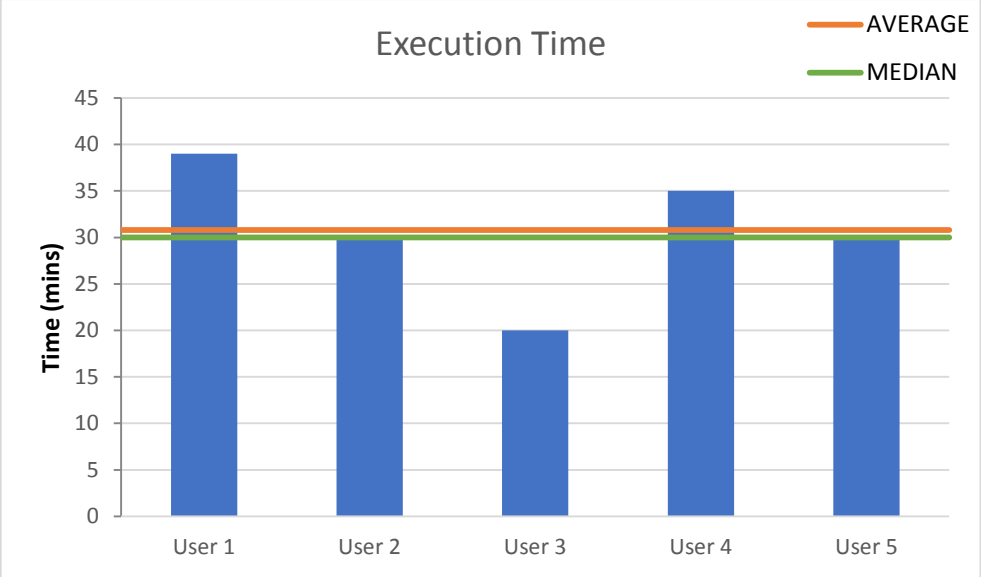

Figure 30 - Execution Time - Scenario 2

What we can observe in Figure 29, is that time almost doubled on the increase of the complexity of the scenario. Users also have different results on this diagram, for instance user 3 managed to execute the scenario in exactly half the time user 1 did. This could be a result of insufficient training of the participants regarding the testing environment. In addition, it indicates that the system should become simpler to use. The first assumption becomes even more supported by the fact that most users asked questions regarding the process, as shown in Figure 15.

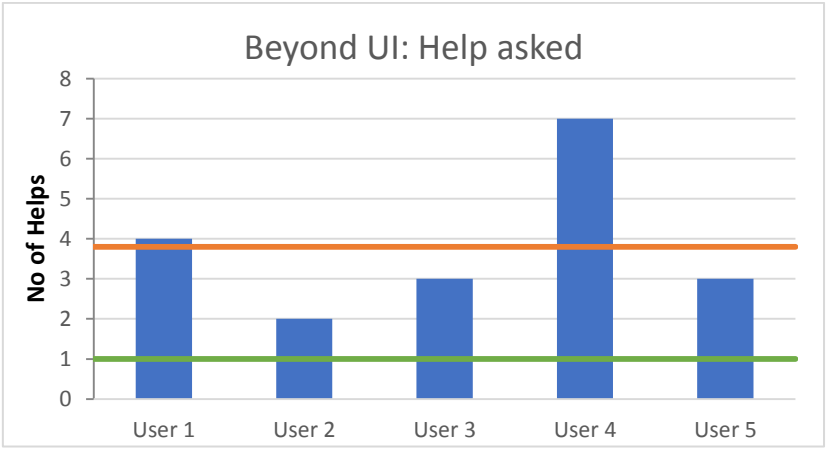

Figure 31 - Help asked, regarding process/ Beyond UI - Scenario 2

Figure 30 shows that all the users asked for help repeatedly regarding the whole process, thus suggesting that the whole procedure needs improvements and that the user needed more assistance and clarifications as the requirements for more complex

testing composition increased. We can see though that their number increased linearly, as time also did, which means that the relation between requirements of test composition, errors and time spent for its writing is linear and not exponential. In addition, users did not make any errors regarding the process/beyond UI part. This is an indication that the testing system should provide by itself more clarifications regarding its use, clarifications that were provided to the participants during the evaluation process as assistance. Such clarifications include but are not limited to the mental model that the API of the artifacts provides, but also a more clear suggestion mechanism regarding the auto-completion feature, especially for object properties suggestion.

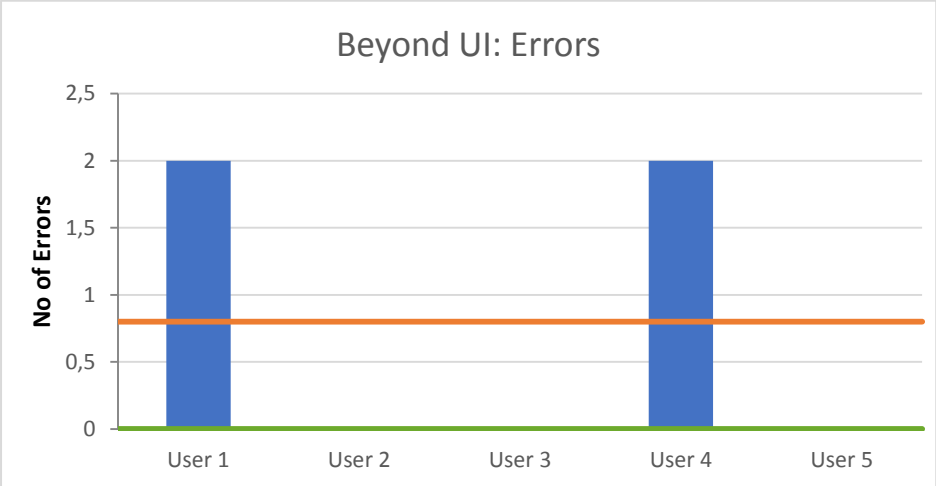

Figure 32 - Process/Beyond UI Errors - Scenario 2

In Figure 31, we can observe that only two out of five users made errors regarding the process of scenario execution. These errors are linked to the mental model regarding the representation and the clarification of the roles of artifacts within the ecosystem, leading them to wrong selections and usage – for instance, one user was confused regarding whether or not he should use a motion detection artifact regarding motion between rooms, or the motion sensor of the house overall. The system needs to clarify the roles of its artifacts so that they are displayed to the user without leaving any margin for confusions and error.

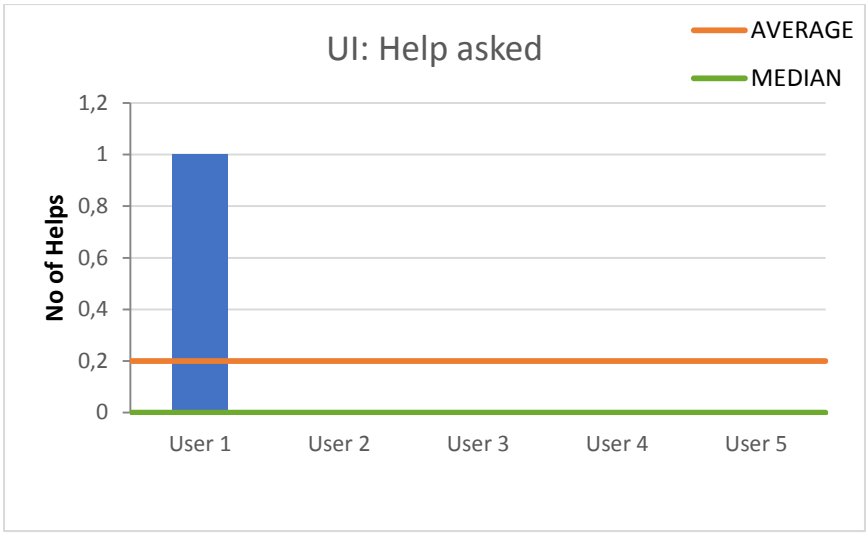

Figure 33 - UI - Help asked - Scenario 2

Regarding errors in this scenario, as shown in Figure 32, only one user was confused with a particular error. The error was related to the numbering options provided by the wizard of the application and their activation and deactivation.

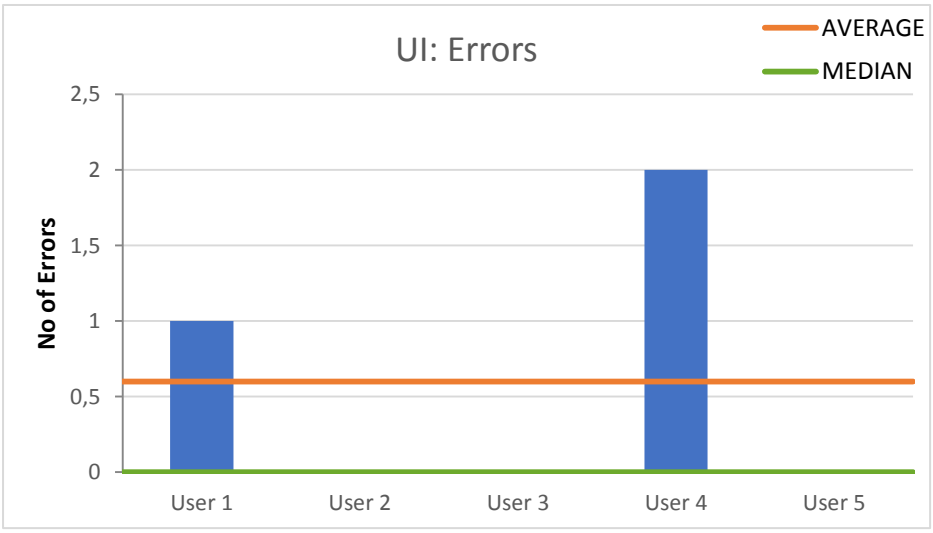

Figure 34 - UI Errors - Scenario 2

Regarding errors in the usage of the UI, as shown in Figure 33, we can observe that two out of five users made a small number of errors, a fact that suggests consideration for improvements of the UI. Those errors were related to the selection of mock components and navigation issues, and users also expressed their concerns about those issues that were considered for the improvement of the system

.

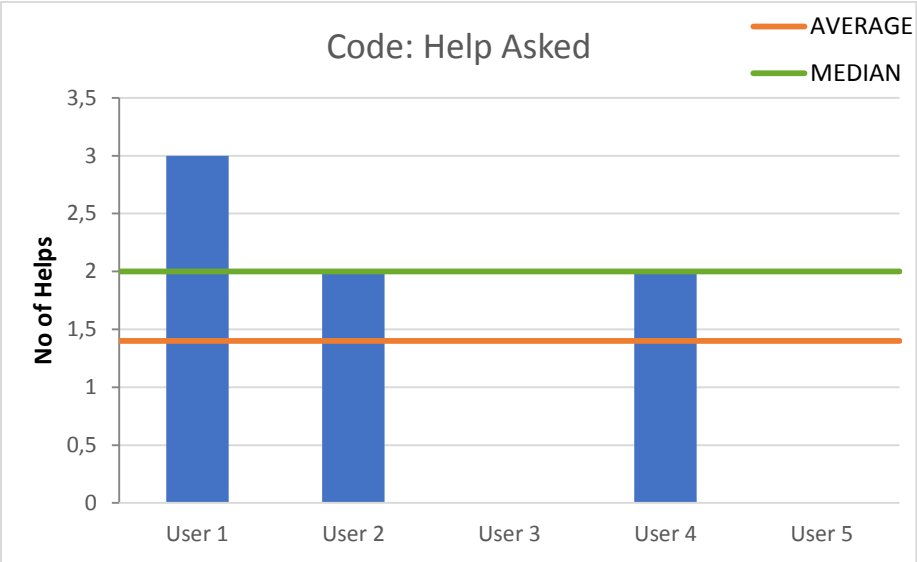

Figure 35 - Code: Help Asked - Scenario 2

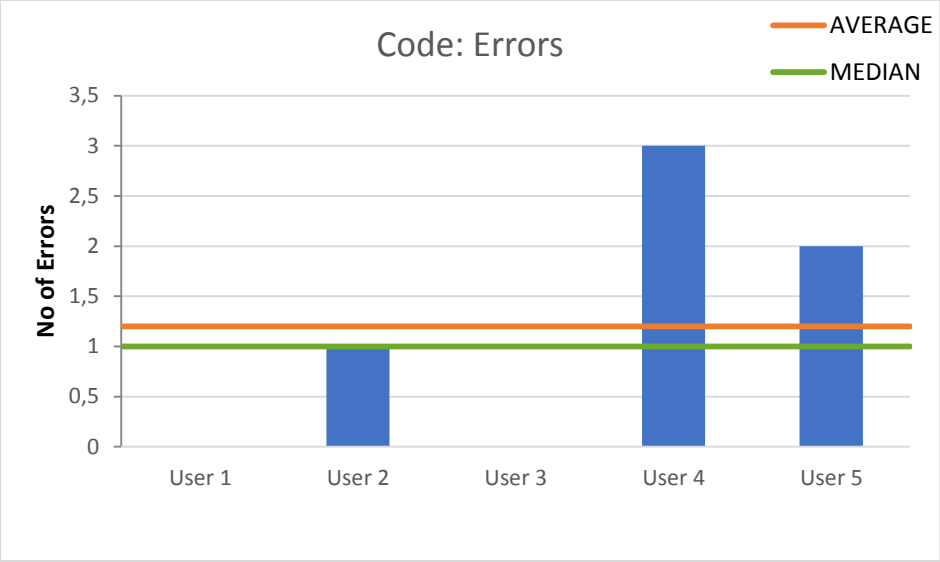

Figure 36 - Code Errors - Scenario 2

Considering that all the users who participated in the evaluation were developers, their requests for help and the errors they would make regarding the composition of code is interesting in terms of observation, as shown in Figures 34 and 35. In general, the amount of help requests and errors related to code was small and mainly had to do with the mental model of the APIs provided to the developers. Regarding help, one common suggestion was the usage of `await`, which some users omitted occasionally. One common case was the misuse of some of the defined event functions (such as `onMotionDetected(callback)`) with its event simulation function (such as `triggerOnMotionDetected()`), about which the participants asked clarifications

in order to use them effectively. Such requests indicated the need for improvements of the system, along with the provision of a more effective API presentation mechanism.

4.3. Scenario 3

After scenario 2, the participants were prompted to execute an already prepare scenario that resulted into the indication of a faulty case into the system, in order to validate whether or not they will be able to identify this error (a problematic lamp that was delaying on its activation). In the end, all users identified the problem successfully.

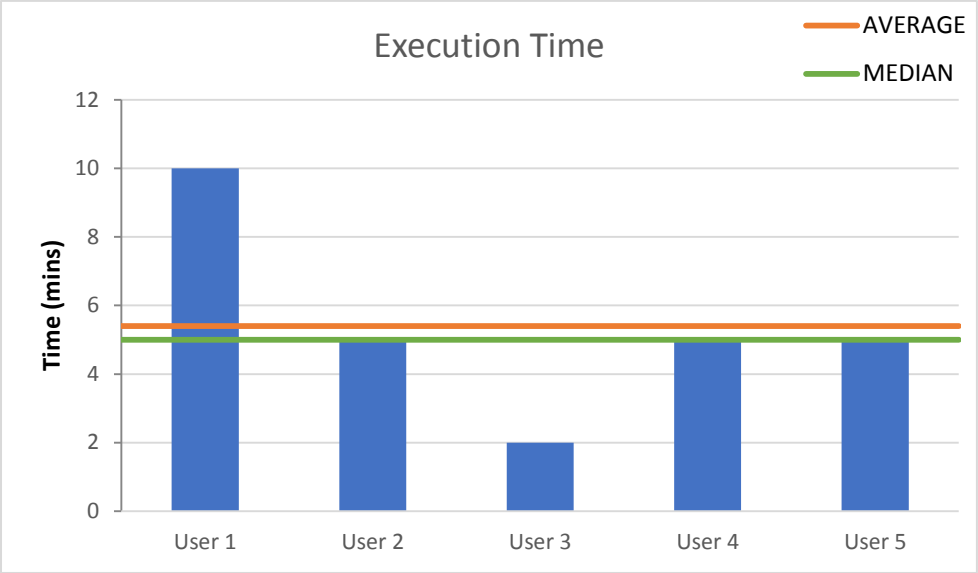

Figure 37 - Execution Time - Scenario 3

Considering that this was a simple case, the execution time was limited to an average of about 5 minutes, as shown in Figure 36, with the users being able to identify the flaw without trouble, in general.

Regarding the process, the users did not make errors as the result they were looking for was straightforward. Two of them asked for some clarifications about what they were searching for, not indicating though any problems on the procedure or the system, as shown in Figure 37.

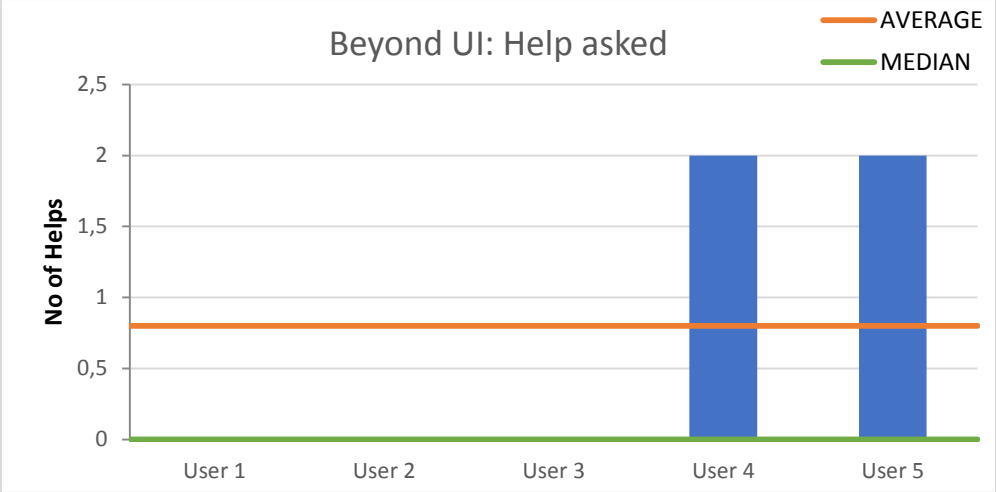

Figure 38 - Beyond UI - Help Asked - Scenario 3

4.4. System Usability Scale

In order to assure that we gain more concise results, we conducted a System Usability Scale [63] test to all five users. Each user was requested to fill the official form of SUS after the execution of the testing scenarios.

Considering that SUS has a usability threshold of 67%, our system gained a score of 83,5%, which indicates that our system was marked as highly usable. It is also worth noting that all users marked the system above 67%.

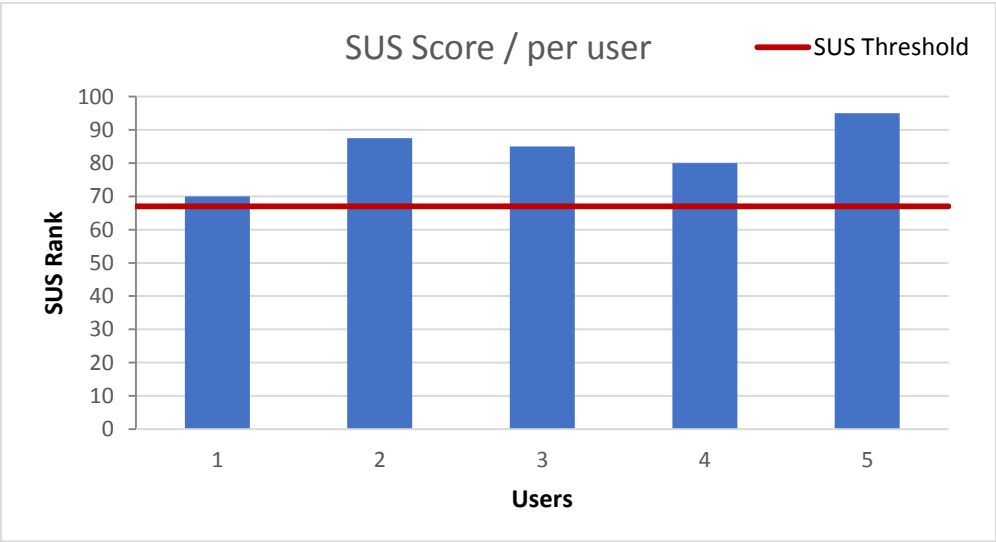

Figure 39 - SUS Score/per user

Making a more specific analysis on SUS, most of the users said they would use the system frequently, that the system was not complex but in general easy to use without the assistance of an IT professional, and that the system functions were well integrated. In addition, they expressed the opinion that the system does not require extensive learning before being used. The results can be observed in Figure 38 below.

4.5. Debriefing and User Comments

After the execution of tests and the SUS questionnaire, the participants were asked some debriefing questions, regarding their opinion on the tool, what they liked the most about the system and what not, and whether they had suggestions about the enhancement of the system. These comments were considered in combination with the comments collected through the whole scenarios execution process via think-aloud.

In general, most of the users found the system a good solution for the testing of AmI environments. Some of them characterized it as "very good", "interesting", "usable" and "very useful with a little practice", similar to their initial mental model of testing such an environment. In fact, the system very close to what they mentioned in the pre-test phase about how they would test an AmI Environment. Many of the participants stated that they liked a number of features of the system, such as the auto completion capabilities – a feature used extensively by all the participants, as well as the stubbing capabilities of components. They also appreciated that they don't have to deal with the complexity of the system. Most of the participants mentioned they would use the system extensively.

Regarding indications and suggestions, the participants proposed a more thorough presentation of the artifacts inside the system, but also a more clear presentation of the API of the artifacts used in the coding part and a more IDE-like User Interface, along with a well-written documentation. In addition, they indicated a number of flaws regarding the UI, such as the indication of component and behavioral script selections which was not found clear by some users, and the navigation system of the wizard.

In addition, the participants suggested improvements regarding the code composition process, such as the usage of `await` in a way that it cannot easily be omitted by mistake and a better component pre-filling mechanism in the code. Some users also recognized a pattern in the test composition process (artifacts selection, triggering of

actions, assertions) and proposed the automatic detection of similar tests for reuse purposes.

Regarding what the participants disliked the most, they indicated that they would like to have a more complete auto completion mechanism, but also a more verbose reporting system. One user indicated that a core missing feature of the system is a debugging mechanism on the tests execution, along with the ability of putting multiple behavioral scripts under test in parallel. In addition, the participants suggested a better test outline mechanism, as this was proven confusing to some of them. Another important suggestion that users made is that they did not know the waiting time they should set in order to test a triggering behavior, which is an important consideration for the system.

In general, the system was well-received and was appraised for its usability and ease to use, hiding large complexity from the developer, achieving its aims with great success.

5. FUTURE WORK

Considering that AmI Environments are scalable and complex systems, their validation process should also not be limited but be extensive and adaptable. In its current state, AmITest currently mainly focuses on the integration testing and provision of basic component simulation capabilities. Our aim is to extend it to a large, sophisticated testing suite, which will provide extended testing capabilities in terms of simulation, integration testing, but also end-to-end testing, and the automatic detection of problematic behaviors inside the ecosystem, along with the provision of error correction recommendation mechanisms, that will contribute to the stability and the long-living of the AmI ecosystem, along with valuable statistical data regarding its behavior. Our aim is to eventually create a full Testing suite that also plays the role of a test automation assistant to the tester at the same time, actively acting for the benefit of the optimal operation of the AmI Ecosystem.

- **Security Measures:** Considering that AmITest applies validation checks to an Ambient Intelligence environment, an environment that collects data from human inhabitants, often raising ethical concerns about privacy and security as a concept, our goal is to preserve any data that the user offers to the system with his/her will, thus not allowing their access to malicious users. To this end, we plan to comply with any security standards applying to the context of specific Ambient Intelligence environments. In addition, we plan to use state-of-the-art security technologies in order to preserve the usage and access of the data of the Ambient Intelligence environment, such as the usage of SSL, OAuth 2.0 and token authentication to our APIs etc.
- **Actual Ecosystem Events Recording and Replay:** One of the first planned extensions concerns the events simulation used for validation purposes. Currently, the system supports the simulation of single events inside the system. While such mechanism proves to be powerful in terms of testing and validation, it can become unrealistic at times in comparison with the actual behavior of a human inhabitant inside the ecosystem. Therefore, we consider the extension of AmITest with an event recording mechanism, providing the ability to the tester to perform actions

inside the actual ecosystem, record those actions in the form of events happening inside the ecosystem in relevance with time. After the recording of actions, the tester will be given the ability to modify, if desired, via provided basic editing capabilities, and eventually replay the original or modified sequence of events. Using such mechanism, the tester will be able to test the validity of a behavioral script along with the optimal behavior of artifacts inside the ecosystem, while simulating an actual user behavior inside the system. This is based on the simple behavioral pattern of checking a device in normal live by use, to check if it behaves well. Via this event mechanism, this can be easily achieved. In fact, our system contains all the tools for the creation of such mechanism with ease.

- **Continuous Integration (CI):** As we mentioned earlier, AmITest is an integral, core part of the AmI Solertis Development and Orchestration studio. In such context, the application of testing can be proved very effective if it becomes integrated in the various phases of the actual AmI ecosystem behavioral development, such as artifacts registration and deployment, along with behavioral code composition and deployment in AmI Solertis. We plan to extend our system in order to provide Continuous Integration capabilities to Solertis. Technically, this will be achieved in combination with a very popular and frequently used Continuous Integration tool, named Jenkins [64]. Jenkins is a tool that aims to provide powerful unit and integration testing capabilities at various phases of the development of software. Such CI tools prove to be very powerful for the application of the automatic unit, integration and end-to-end testing. Considering that our system is based on the concepts of unit and mainly integration testing and that the development process consists of a number of phases, we consider that such an addition will be highly beneficial for the validation of the behavior of AmI environments, developed and orchestrated via AmI Solertis.
- **Extended Visual Programming Capabilities:** Currently the visual programming capabilities provided by AmITest are based on the Google Blockly platform, mainly providing test composition capabilities via jigsaw puzzles. Considering that Visual Programming (VP) is an area of separate ongoing research, we aim to constantly provide better mechanisms for Visual Programming. Our future work plans include the extension of the Blockly platform, in order to provide jigsaw

composition assistance capabilities, similar to auto completion and blocks suggestion, via the usage of popup windows.

- **3D Modeling Simulation Capabilities:** In the related work section, we observed that systems such as InSitu, UBIWISE and UbiREAL focus on the simulation of Smart Environments via the usage of 3D models, using various Graphic Engines such as Quake III Arena Graphics Engine and the Half Life 2 engine. We plan to further explore this area and integrate state-of-the-art technologies related to VP inside our system. Those plans include the consideration of tools that virtually represent the AmI environment, via 2D or 3D models, in order to simplify the interaction for novice end users.
- **Log Inferencing & Analysis:** The ultimate aim of AmITest is to provide validation mechanisms to the testers of the AmI ecosystem, but also actively contribute to the prevention of errors and the suggestion of fixes in cases of flaws or deficiencies inside the ecosystem. Toward this end, we plan the implementation of a mechanism that will use information collected via the logging mechanisms of AmI Solertis, and apply sophisticated pattern recognition algorithms in order to infer possible deficiencies inside the ecosystem, produce reports and suggest solutions to the users of the system, or the system itself, assisting to the operation of its potential environment automatic repairing mechanisms. This includes the analysis of reports already generated by AmITest via unit and integration testing analysis, such as the reports generated during the execution of the scripts composed in the Test Composition Wizard. AmITest produces such results in a well specified, well-structured format (JSON), well understandable and widely used for logging and configuration.
- **More Verbose Reporting System:** The evaluation experiment showed that there is a need for a more verbose reporting system. We plan to enhance our existing reporting system with the provision of more information regarding the tests execution.
- **Extended Debugging Capabilities:** Another very important result of the evaluation experiment is that there is a need for a more sophisticated debugging mechanism regarding the test execution, such as step-by-step execution. We plan to integrate such a mechanism into our system.

Through adding such features to the system, AmITest will become much more than just a testing tool. The aim is to eventually create a full testing suite that also plays the role of a test automation assistant to the tester at the same time, actively acting for the benefit of the optimal operation of the AmI ecosystem, and eventually the benefit of its human inhabitants.

6. CONCLUSIONS

This work has described AmITest, a testing framework for Ambient Intelligence Environments targeted to check the validity of operations programmed by end users in a Smart Environment, which is a core, integral component of AmI Solertis, a system for the orchestration and development of Ambient Intelligence systems.

The framework provides mechanisms for simple, straightforward, yet powerful and effective testing of the functionality of technological artifacts inside a Smart Environment. In addition, the framework provides an optional sandboxing execution environment, in order to apply testing assertions without affecting the actual AmI ecosystem, thus providing mechanism for its effective and extensive testing regardless of whether or not a behavioral definition (script) has been deployed and executed inside the Smart Environment. Finally the system generates and provides reporting information to the tester and AmI Solertis, the whole AmI environment orchestration system.

AmITest ultimately targets at both programming and non-programming professional users, and supports testing via scripting and Visual Programming. Even though well-established user-friendly visualization techniques have been currently applied (e.g., Wizards, Blockly etc.), following the iterative approach of the User-Centered Design (UCD) process[65][66], both users and experienced developers of AmI servicers will be actively involved in the design process of the visual tools, through preliminary evaluation sessions and participatory design sessions, to maximize their usability for both groups. Regarding the beta version of our system, we have conducted a user evaluation with focus on users with IT expertise, and we presented the results. Upon the release of version 1.0 of AmITest, we plan to conduct an extensive in-vivo full-scale evaluation experiment both with HCI experts and users in order to examine and improve the usability of the AmITest editing facilities.

7. REFERENCES

[1] Augusto, Juan, and Paul Mccullagh. "Ambient Intelligence: Concepts and applications." *Computer Science and Information Systems* 4.1 (2007): 1-27. Web.

[2] J. Krumm, Ubiquitous Computing Fundamentals. Boca Ragon: Chapman & Hall/CRC Press, 2010.

[3] F. Adelstein, Fundamentals of Mobile and Pervasive Computing. New York: McGraw-Hill, 2005.

[4] Leonidis, Asterios, Margherita Antona, and Constantine Stephanidis. "Enabling programmability of smart learning environments by teachers." International Conference on Distributed, Ambient, and Pervasive Interactions. Springer International Publishing, 2015.

[5] Leonidis, Asterios. AmI Solertis: an Online Platform that Facilitates the Definition of "Smart" Behaviours in AmI Environments. PhD Dissertation, University of Crete, forthcoming.

[6] "Internet of Things Global Standards Initiative." *ITU*. N.p., n.d. Web. 03 Apr. 2017.

[7] Aarts, E. H. L., and José Luis. Encarnação. *True visions the emergence of ambient intelligence*. Berlin: Springer, 2008. Print.

[8] Danado, Jose, and Fabio Paternò. "Puzzle: A Visual-Based Environment for End User Development in Touch-Based Mobile Phones." *Human-Centered Software Engineering Lecture Notes in Computer Science* (2012): 199-216. Web.

[9] "VPL Introduction - msdn.microsoft.com." N.p., n.d. Web.

[10] Chen, Yinong, and Gennaro De Luca. "VIPLE: Visual IoT/Robotics Programming Language Environment for Computer Science Education." *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016): n. pag. Web.

[11] Dahl, Yngve, and Reidar-Martin Svendsen. "End-User Composition Interfaces for Smart Environments: A Preliminary Study of Usability Factors." *Lecture Notes in Computer Science Design, User Experience, and Usability. Theory, Methods, Tools and Practice* (2011): 118-27. Web.

[12] Zapier. "Connect Your Apps and Automate Workflows." *The best apps. Better together. - Zapier*. N.p., n.d. Web.

[13] "IFTTT." *Learn how IFTTT works - IFTTT*. N.p., n.d. Web.

[14] Nishikawa, Hiroshi, Shinya Yamamoto, Morihiko Tamai, Kouji Nishigaki, Tomoya Kitani, Naoki Shibata, Keiichi Yasumoto, and Minoru Ito. "UbiREAL: Realistic Smartspace Simulator for Systematic Testing." *Lecture Notes in Computer Science UbiComp 2006: Ubiquitous Computing* (2006): 459-76. Web.

[15] Duvall, Paul M., Steve Matyas, and Andrew Glover. *Continuous integration improving software quality and reducing risk*. Upper Saddle River, NJ: Addison-Wesley, 2013. Print.

[16] Newman, Sam. *Building Microservices*. N.p.: O'Reilly, 2015. Print.

[17] Crispin, Lisa, and Janet Gregory. *Agile testing a practical guide for testers and agile teams*. Upper Saddle River: Addison-Wesley, 2014. Print.

[18] I. Georgalis, Y. Tanaka, N. Spyratos, and C. Stephanidis, Programming Smart Object Federations for Simulating and Implementing Ambient Intelligence Scenarios. In C. Benavente-Peces and J. Filipethe (Eds.), Proceedings of the

3rd International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2013), Barcelona, Spain, 19-21 February 2013, pp. 5-15. Portugal: SCITEPress.
[19] *Swagger Specification*. N.p., n.d. Web. <http://swagger.io/specification/>.
[20] *JavaScript Programming Language*. N.p., n.d. Web. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
[21] *JavaScript Promise*. N.p., n.d. Web. <https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise>.
[22] *JavaScript async function* . N.p., n.d. Web. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function>.
[23] *JavaScript await*. N.p., n.d. Web. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>.
[24] *ECMAScript Language Specification*. N.p., n.d. Web. <https://www.ecma-international.org/publications/standards/Ecma-262.htm>.
[25] *Redis PubSub*. N.p., n.d. Web. <https://redis.io/topics/pubsub>.
[26] Tragos, Elias, Alexandros Fragkiadakis, Vangelis Angelakis, and Henrich C. Pöhls. "Designing Secure IoT Architectures for Smart City Applications." *Designing, Developing, and Facilitating Smart Cities* (2016): 63-87. Web.
[27] Louloudakis, N., Leonidis, A., & Stephanidis, C. (2016). AmITest: A Testing Framework for Ambient Intelligence Learning Applications. In the *Proceedings of the Eighth International Conference on Mobile, Hybrid, and On-line Learning (eLmL 2016)*, Venice, Italy, 24-28 April.
[28] Lu, Heng. "A Context-Oriented Framework for Software Testing in Pervasive Environment." Proc. of 29th International Conference on Software Engineering. N.p.: IEEE, n.d. N. pag. Print.
[29] Satoh, I. "Software testing for mobile and ubiquitous computing." *The Sixth International Symposium on Autonomous Decentralized Systems, 2003. ISADS 2003.* (n.d.): n. pag. Web.
[30] J.J. Barton, V. Vijayaraghavan, UBIWISE, a simulator for ubiquitus computing systems design, Tech Rep. HPL-2003-93, Mobile and Media Systems Laboratory, Hewlett Packard Laboratories Palo Alto (2003)
[31] Jouve, Wilfried, Julien Bruneau, and Charles Consel. "DiaSim: A parameterized simulator for pervasive computing applications." *2009 IEEE International Conference on Pervasive Computing and Communications* (2009): n. pag. Web.
[32] *UPNP Specification*. N.p., n.d. Web. <https://openconnectivity.org/resources/specifications/upnp/specifications>.
[33] O'Neill, Eleanor, Owen Conlan, and David Lewis. "Situation-based testing for pervasive computing environments." *Pervasive and Mobile Computing* 9.1 (2013): 76-97. Web.
[34] Martín, Diego, Ramón Alcarria, Álvaro Sánchez-Picot, and Tomás Robles. "An Ambient Intelligence Framework for End-User Service Provisioning in a Hospital Pharmacy: a Case Study." *Journal of Medical Systems* 39.10 (2015): n. pag. Web.
[35] Ferrandez-Pastor, Francisco Javier, Juan Manuel Garcia-Chamizo, Mario Nieto-Hidalgo, and Francisco Florez-Revuelta. " DAI Virtual Lab: a Virtual Laboratory for Testing Ambient IntelligenceDigital Services." (n.d.): n. pag. Web.
[36] Petrova-Antonova, Dessislava, Sylvia Ilieva, and Denitsa Manova. "TASSA: A Testing as a Service Framework for Web Service Compositions." *Proceedings of*

the International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn (2016): n. pag. Web.
[37] Whitley, K. N., and Alan F. Blackwell. "Visual programming: : The Outlook from Academia and Industry." *Papers presented at the seventh workshop on Empirical studies of programmers - ESP '97* (1997): n. pag. Web.
[38] M. Resnick et al, "Scratch: Programming for All." Communications of the ACM Commun. ACM 52, no. 11 (2009), pp. 60-67.
[39] O. Gray and M. Young, 2007. "Video Games: A New Interface for Non-Professional Game Developers". In ACM International Conference on Computer-Human Interaction (CHI 2007), USA: San Jose.
[40] N. Tillmann, M. Moskal, J. De Halleux, and M. Fahndrich, "TouchDevelop: Programming Cloud-connected Mobile Devices via Touchscreen." Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - ONWARD '11, 2011, pp. 49-60.
[41] D. Wolber, "App Inventor and Real-world Motivation." Proceedings of the 42nd ACM Technical Symposium on Computer Science Education - SIGCSE '11, 2011, pp. 601-606.
[42] "Blockly | Google Developers." Google Developers. Accessed January 25, 2016. https://developers.google.com/blockly/.
[43] B. Waldie, Automator for Mac OS X 10.6 Snow Leopard. Berkeley, CA: Peachpit Press, 2010.
[44] R. Ierusalimschy, Programming in Lua. Rio De Janeiro: Lua.org, 2006.
[45] W. Goldstone, Unity 3.x Game Development Essentials: Game Development with C♯ an*d Javascript. Birmingham, UK: Packt Publishing, 2011.*
[46] R. J. Cox, and P. S. Crowther, "A Review of Linden Scripting Language and Its Role in Second Life." Lecture Notes in Computer Science Computer-Mediated Social Networking, 2009, pp. 35-47.
[47] "CodeMirror." *CodeMirror*. N.p., n.d. Web. 03 Apr. 2017. <http://codemirror.com/>.
[48] "MochaJS" *MochaJS*. N.p., n.d. Web. <https://mochajs.org/>.
[49] "ShouldJS" *ShouldJS*. N.p., n.d. Web. < https://shouldjs.github.io/>
[50] "Chai" *Chai*. N.p., n.d. Web. <http://chaijs.com/>.
[51] "Jasmine Documentation." *Jasmine*. N.p., n.d. Web. 03 Apr. 2017. <http://jasmine.github.io/>.
[52] "AngularJS - Superheroic JavaScript MVW Framework." *AngularJS - Superheroic JavaScript MVW Framework*. N.p., n.d. Web. 03 Apr. 2017.
[53] Grove, Ralph F., and Eray Ozkan. "The MVC-Web Design Pattern." *Proceedings of the 7th International Conference on Web Information Systems and Technologies* (2011): n. pag. Web.
[54] *" Handlebars.js" Handlebars.js:*. N.p., n.d. Web. 03 Apr. 2017. <http://handlebarsjs.com/>.
[55] "Tern." *Tern*. N.p., n.d. Web. 03 Apr. 2017. <http://ternjs.net/>.
[56] "Introducing JSON." *JSON*. N.p., n.d. Web. 03 Apr. 2017. <http://www.json.org/>.
[57] "JSDoc." *JSDoc*. N.p., n.d. Web. 03 Apr. 2017.
[58] "Standalone test spies, stubs and mocks for JavaScript. Works with any unit testing framework." *Sinon.JS*. N.p., n.d. Web. 03 Apr. 2017. <http://sinonjs.org/>.

[59] Zhao, Li, R. Iyer, S. Makineni, and L. Bhuyan. "Anatomy and Performance of SSL Processing." *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* (2005): n. pag. Web.
[60] Jones, M., and D. Hardt. "The OAuth 2.0 Authorization Framework: Bearer Token Usage." (2012): n. pag. Web.
[61] Nielsen, Jacob. "Success Rate: The Simplest Usability Metric." *Success Rate: The Simplest Usability Metric*. N.p., n.d. Web. 03 Apr. 2017. <https://www.nngroup.com/articles/success-rate-the-simplest-usability-metric/>.
[62] Olmsted-Hawala, Erica L., Elizabeth D. Murphy, Sam Hawala, and Kathleen T. Ashenfelter. "Think-aloud protocols: Analyzing three different think-aloud protocols with counts of verbalized frustrations in a usability study of an information-rich Web site." *2010 IEEE International Professional Comunication Conference* (2010): n. pag. Web.

[63] Brooke, John. "SUS - A quick and dirty usability scale." (n.d.): n. pag. Web.\
[64] "Jenkins Automation Server." *Jenkins*. N.p., n.d. Web. 03 Apr. 2017. <https://jenkins.io/>.
[65] Norman, Donald A. *The design of everyday things*. New York: Doubleday, 1990. Print.
[66] Lowdermilk, Travis. *User-centered design*. Place of publication not identified: Shroff Publishers & Distr, 2013. Print.