AmITest: A Testing Framework for Ambient Intelligence Learning Applications

Nikolaos Louloudakis, Asterios Leonidis and Constantine Stephanidis
Foundation for Research and Technology – Hellas (FORTH) - Institute of Computer Science
N. Plastira 100, Vassilika Vouton, GR-700 13
Heraklion, Crete, Greece
e-mail: {luludak, leonidis, cs}@ics.forth.gr

Constantine Stephanidis
University of Crete, Department of Computer Science
Heraklion, Crete, Greece
e-mail: cs@csd.uoc.gr

***Abstract*— As the Ambient Intelligence (AmI) paradigm emerges and develops, applications in education are attracting increasing attention. For maximum educational efficiency, extensiveness and adaptation to the needs of their users, AmI systems in education need to be easily programmable. Considering 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 this paper, we propose AmITest, a framework that effectively allows the testing and the validation of behavioral programs written by users in a simple, yet direct and effective way. AmITest is part of a complete end-user development suite named AmIClass, which allows the effective programming of AmI educational environments by non-computer professionals.**

Keywords-visual programming; end-user testing; ubiquitous environments; smart learning environment; ambient intelligence testing

I. 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] and Pervasive Computing [2] that 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), ubuquitous 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 spare 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. Such a domain with promising potentials is the domain of education, particularly regarding the concept of Just-in-Time Learning [3] (i.e., the provision of learning material at the right time, the right place and the appropriate format).

An Intelligent environment that promotes learning by offering to its users educational material when and where they are ready to espouse knowledge the most, based on the current task at hand and the available technological artifacts that could be used in order to make the information as easily adoptable as possible (e.g., sensors, actuators, interactive devices, etc.), is called a Smart Learning Environment (SLE) [4].

A key requirement of SLEs is that their behavior and interaction policies need to be easily programmed by their end-users, who most likely will not be computer professionals. In this respect, Leonidis et al [5] have proposed AmIClass, a framework for end-user visual programming aiming primarily to users that are non-computer professionals. An important aspect 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 paper proposes the AmITest Framework, a testing framework for AmI environments, with its main focus on SLEs.

AmITest aims to support the users that define the behavior of the SLE (i.e., SLE programmers), rather than the end-users of the SLEs themselves (e.g., teachers, students, school principals, etc.). The programming expertise of AmITest users however may vary greatly ranging from motivated teachers 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 learning facilities. To accommodate both groups, while considering that the majority of the target users will not be experts, AmITest makes the testing process as straightforward as possible, by offering a visual programming tool for the end-users to define their tests. The AmITest framework is a novel part of the AmIClass Framework [5] that provides testing capabilities using the dynamic scripting language of AmIClass in order to test the artifacts of the SLE being tested as subject.

The AmITest framework is mainly consituted of two main components: (1) *The ClassScript Testing Agents*, a framework for testing and validating the behavior of SLEs

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-471-8

defined in an imperative domain-specific language named ClassScript and (2) *The Tests Management and Deployment Suite*, a suite that facilitates end-users to visually program a number of tests regarding the AmI environment behavior using a building block-based Graphical User Interface (GUI). These components interoperate in order to provide the best programming options to the end-users so that they can configure parts of the behavior of the system (e.g., articulate user input for a certain scenario, transcript the behavior of a fictional user, etc.) and validate whether the intended behavior meets the original expectation.

The paper is organized as follows. Section II describes relevant approaches to the AmITest framework. Section III outlines the requirements that should be met to apply successful testing of a SLE. Section IV describes the implementation details. Section V presents a proof-of-concept scenario. Section VI focuses on the biggest challenges met in order for AmITest to do efficient testing, and Section VII concludes the paper by discussing the current status of the framework and discussing a list of potential improvements and additions.

II. RELATED WORK

AmITest focuses on the creation of tests that aim to validate the behavior of an SLE. The behavior itself is defined in AmIClass using the ClassScript language [5], a dynamic, untyped language used specifically for the definition of the behavior of particular artifacts inside an SLE, and macroscopically, the behavior of the SLE as a whole [5]. AmITest introduces a library that enables testing of ClassScript. AmITest is based on similar testing frameworks for untyped languages, and in particular on two testing frameworks for the JavaScript programming language [6], Jasmine [7] and QUnit [8].

As aforementioned, non-computer professionals will mainly be asked to program and test the behavior of an SLE, 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 [9]. This paradigm is effectively used in order to provide programming capabilities to systems that target non-professional users in various systems with diverse objectives. AmIClass [5] enables the definition of the behavior of SLEs via visual programming even by novice users (e.g., teachers). Scratch [10] is a visual programming environment primarily targeted to users in ages between 8 and 16 years old, with limited to none programming experience, that aims to teach them programming while working on meaningful projects such as animated stories and games via a visual programming editor. Virtuoso [11] is a visual tool for creating educational games aiming primarily non-professional users, based on Valve's game engine. TouchDevelop [12] 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 [13] is a platform from MIT which provides a web-based visual programming tool for designing mobile applications online. Automator [14] is a visual scheduling tool providing capabilities of repetitive automation tasks in the Mac OSX platform.

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, only AmIClass 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 an SLE, but also offers both visual and script editing facilities to accommodate users with different levels of expertise. Consequently, any user will be able to program her own test cases and test the behavior of the SLE easily. 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 [15] and JavaScript [6] having played an important role in the widespread adoption of extensible game engines (such as Unity [16]), and even further, to the introduction of games that players can freely customize (e.g., the game "Second Life" offered the Linden Scripting Language [17] through which players were able to create in-game elements).

III. FRAMEWORK REQUIREMENTS

Considering that SLEs 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 SLE behavior as a whole.

In order for the proposed system to efficiently test the functionality of each individual artifact and the behavior of the SLE as a whole, each artifact should have installed a lightweight service, the *ClassScript Testing Agent* (CTA), which facilitates the orchestrator of the testing operations done in the artifacts, and works as the delegate of the *Tests Management and Deployment Suite (TMDS),* which is responsible for the definition of the testing actions on each artifact. The CTAis practically a service communicating with the Service Mediator Agent of the AmIClass framework and is responsible for the installation, deployment and execution of the test scripts on that artifact.

Each artifact should also implement a lightweight Application Programming Interface (API) called *ISchoolArtifact* in order to allow the system execute certain operations necessary for testing, such as requesting the form of the information provided from each artifact (i.e., the information schema), accessing that information to determine the status of the artifact, etc,.

As aforementioned, artifacts of an SLE interoperate with each other in a distributed manner [18], as they are different remote sub-systems that coexist inside the SLE. To satisfy the increased communication needs stemming from both the "normal" SLE operation and the testing purposes, the proprietary FAmINE middleware [19] was used. The *ISchoolArtifact* interface is defined as a FAmINE component and via that interface the distributed systems can communicate with each other.

Finally, since the testing framework will be a part of the AmIClass Framework, each artifact should meet the requirements of AmIClass as described in [5].

IV. SYSTEM ARCHITECTURE

The AmITest system is an integral part of the AmIClass suite, thus it follows a similar architectural structure from an engineering perspective. The AmiTest framework consists of: (1) the delegate ClassScript Testing Agents (CTA), which get installed on every artifact and are responsible for the installation and local execution of the testing scripts, and (2) a master web-based suite, the Tests Management and Deployment Suite (TMDS*)*, responsible for the creation, overview and management of the testing procedure. The overall architecture is depicted in Fig. 1 below.

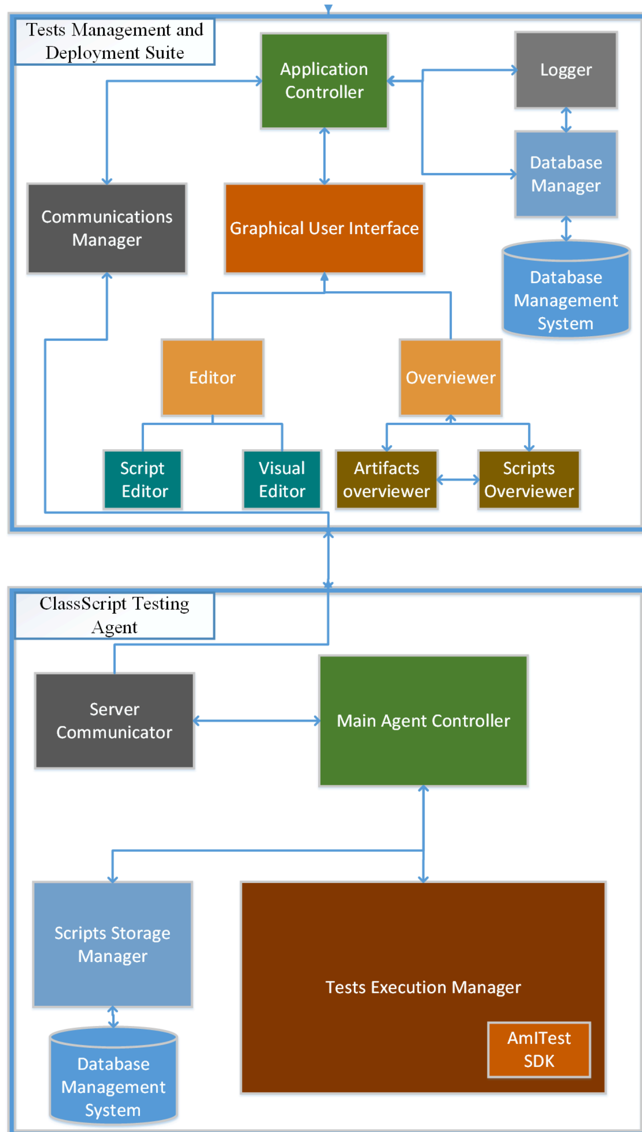

Figure 1. Architecture of the AmITest framework

In Addition, CTA is responsible for managing any locally installed scripts (i.e., update, delete). Before its initial execution, each script is cached locally to minimize its startup time since for every subsequent execution there will be no need to retrieve it from the main repository. The CTA also ensures that the local version is always the latest one, thus on any update it replaces the old version with the new.

The Standard Development Kit (SDK) of AmITest is a native library of the AmIClass framework – part of the language implementation, and the test scripts of the AmITest framework are ClassScript language scripts. Therefore, they are executed by the installed AmIClass interpreter without any modifications. The SDK consists of three specially-purposed APIs: the Artifacts API, the Invoker API and the Tester API.

The Artifacts API handles the information retrieval of the data structures and the actual data of an artifact. Upon launch, it loads the information schema of the underlying artifact, and then periodically collects any data used from the test scripts to validate its behavior and provides them to the Tests Management and Deployment Suite for updating the artifact's overview, enabling test editing, and for logging purposes. Its objective is two-fold: (1) provide the server-side TMDS with information that will assist and enhance the development process of the end-users, and (2) retrieve and use the artifacts data for assertions checking when validating the behavior of a component or of the SLE as a whole.

The Invoker API let programmers schedule valid invocations of a function or triggering of events, in order to check whether the correct behavior has been applied based on the respective "expectations" (i.e., assertions) of the artifact(s) of the SLE. Considering that the Invoker API will mainly focus on the invocation of asynchronous functions, a mechanism that address any dirty object instances existence has been considered. The Invoker API is designed and implemented based on the Promises pattern. A Promise is a pattern which represents the result of an asynchronous, long running and potentially, but not necessarily, complete operation, using an object instance which represents the promised result of the operation. This concept is common on asynchronous programming, and various frameworks implement it for both typed and untyped programming languages, such as C++ Promises [20] and Javascript Promises [21].

The Tester API is responsible for the evaluation of any assertions, named Expectations, relevant to that artifact or the overall SLE. The most common use of Expectations is in combination with Promises, in order to apply checks on the data of one or more artifacts within an SLE: considering that the testing process is an asynchronous task by itself, artifact-specific promises are used to ensure that expectation checking on the artifacts will be performed when the objects are in a ready, clean state and not before. Such an example is depicted in Fig. 2; when all the necessary events are handled by the respective components -the promises are satisfied-, only then the expectations will be evaluated. The Tester API offers a variety of checking options, such as numerical and string checking, shallow and deep object equality comparison, regex checking, etc.

All the components of a CTA are orchestrated effectively via the Main Agent Controller, which orchestrates all the

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-471-8

components aforementioned for a proper functionality. The controller is responsible to enable all the components required for any actions needed to be done, such as script installation, script execution, server briefing about the scripts and the artifact status etc.

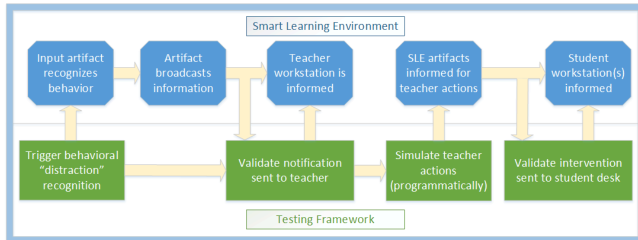

Figure 2. Flow of Information across core SLE (blue boxes) and AmITest (green boxes) components

A. *The Tests Management and Deployment Suite*

The Tests Management and Deployment Suite (TMDS) is responsible for the management of all the testing scripts that exist in the SLE. The suite has two main components: the *ClassScript Integrated Development Environment (CIDE)* and the *Management Suite (MS)*. CIDE is an environment for the development of the scripts that define the behavior of the artifact (i.e., its business logic) and testing scripts that will validate it. MSis a Graphical Suite that facilitates the overview, remote installation, supervision and execution of any testing scripts and consists of the following components: the Application Controller, the Graphical User Interface, the Database Manager, the Logger and the Communications Manager.

The Application Controller is the main controller of the server-side application, which orchestrates all the operations of every component with respect to the testing procedure. The Graphical User Interface is the main Graphical Component of the system, and consists of the Test Scripts Editor and the Overviewer components.

The Test Scripts Editor provides the end-users with an environment where they can create new scripts from scratch or edit existing ones, either visually or via textual scripting since both modes are interoperable. In Visual Editing Mode, the end-user can create and/or modify existing tests using a visual tool of building blocks (based on a Google's Blockly project [22]), as depicted in Fig. 3. This mode is more suitable for users with very little or no programming experience giving them the capability of creating effective test scripts. This tool will eventually generate valid ClassScript code (as shown in Fig. 4) and any artifact will be able to execute it directly using its installed ClassScript interpreter. In Scripting Mode, the end-user can write the tests for the application directly in the ClassScript language, using an integrated WYSIWYG text editor. This mode increases the expressiveness of the scripting tests, it is considered to be more difficult for novice users, but more powerful for users with some programming experience. The system though attempts to assist the end-user programming as much as possible, providing auto-completion capabilities along with syntax highlighting capabilities.

The Overviewer is the graphical tool that presents the overview of the Artifacts and the Scripts existing in the SLE. In particular, apart from their aggregated statistics, for each SLE it provides an overview of the installed, deployed and/or currently executing behavioral and testing scripts. This component also consists of two smaller components, the Artifacts Overviewer, responsible for the overview and management, from a testing perspective, of the various artifacts present in an SLE (e.g., activation and deactivation of facilities) and the Scripts Overviewer, responsible for the overview and the management of the scripts installed in the main repository, or cached locally in every artifact. The end-users can use this component in order to install more scripts on each artifact, but also to monitor a script's execution, its outcomes or even interact with it in real-time (e.g., inspect its status, breakpoint or stop it etc.).

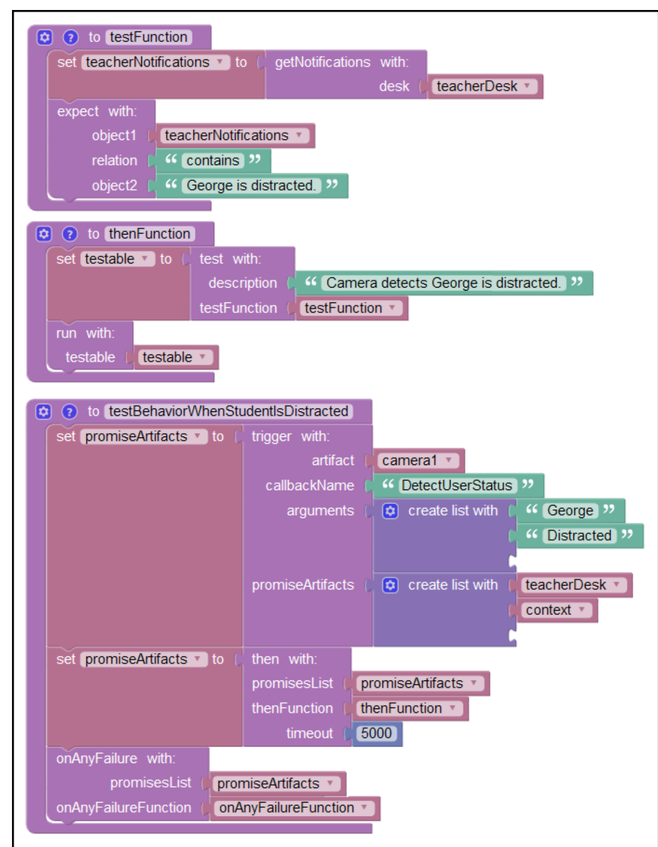

Figure 3. Definition of a test using AmITest Visual Editor

TMDS also contains the Database Manager component for the manipulation of the database operations regarding the test scripts and any relevant information about them (e.g., logging information, information schema, history, etc.). This component is responsible for all the Create-Read-Update-Delete (CRUD) operations of the Database Management System (DBMS) held in the server where the application is installed. The Logger is a component responsible for managing the logging operations, and works in close collaboration with the Database Manager component.

Copyright (c) IARIA, 2016.

Finally, the suite contains the Communications Manager, a component for the two-way communication between the artifacts and the Suite on the server. This component receives all the data needed in order to inform the Suite Graphical User Interface, logging data, scripts data, etc., and propagates any commands to the artifacts including user-oriented commands (e.g., resulting from the interacting with the Suite, such as new scripts installations, scripts executions on an artifact, etc.) or system-oriented commands (e.g., predefined reaction of a fictional user to a certain event, etc.).

V. PROOF OF CONCEPT

A. Test Case Scenario

AmISchool [23] is an ambient educational test bed which consists of a number of artifacts: four touch-enabled AmIDesks [24][25], an interactive teacher workstation, a large projection screen, a Heating, Ventilation, and Air Conditioning (HVAC) controller, alights control system, a sophisticated vision-based user tracking system [26] and various AmI-oriented learning applications [23][27][28]. It employs the ClassScript framework [5] in order to enable the definition of learning plans (i.e., scripts) that control its behavior by the teachers themselves, thus it provides the context for our test case scenario.

Mrs. Smith is a 42-years old history teacher, with very limited programming experience mostly related to formulas creation in a spreadsheets processor to calculate her students grades, who teaches the 5th grade in that school. After a few unsuccesfull attempts, shehas managed to define and validate the general behavior of the classroom both in terms of physical conditions and privacy; she has created scripts that instruct: (1) the classroom to automatically turn on the lights when students are present and control the room's temperature to maximize students' convenience, (2) the AmIDesks to initiate the login procedure when a student is sitting in front of them and show the contents of her presentation if the relevant application is on the foreground on her workstation.

Today, she wants to use for the first time the available Student Attention Monitoring and Intervention system, which in case of inattention can actively intervene, thus she needs program eventually validate a simple intervention that will aim to motivate a distracted student. Therefore, she instructs the vision component to track the gaze of each individual student and notify the teacher if inattention is detected. Upon notification, she wants to be able to either virtually poke that student or activate a quick educational mini-game to regain attention and increase interest for participation.

B. SLE Behavior Programming and Testing

The SLE behavior is defined using the ClassScript's visual editor, while the AmITest framework facilitates its simulation and validation via the Tests Management and Deployment Suite. In order for Mrs. Smith to validate the behavior described above, she can use the AmITest framework in order to simulate the actions of virtual students in order to and validate if the SLE performs as expected. For that to be achieved, firstly she defines two auxiliary testing blocks (i.e., functions), one that checks whether the teacher's workstation displays a notification when inattention is detected and another that validates that whenever the teacher launches a mini-game in a student's desk, then that game is the only active application (Fig. 3 and Fig. 4 present those functions in the user-friendly visual format and the automatically generated ClassScript code respectively). Afterwards, she creates a virtual student, in order to simulate a student distraction behavior for SLE behavior validation purposes. This virtual student will simulate a distraction after a few seconds in order to trigger the overall detection and reaction process, as depicted in Figure 2.

```
function testBehaviorWhenStudentDistracted() {
  $executor.trigger($camera1, "DetectUserStatus",
                    ["George", "Distracted"],
                    //Artifact promises passed as argument (array).
                    [$deskOfTeacher, $context]).then(function() {

    $tester.test("Camera detects George is distracted.", function() {
      $teacherNotifications = $deskOfTeacher.getNotifications();
      $tester.expect($teacherNotifications).toContain("George is distracted.");
    }).run();

    //Definition of SLE's reaction to student's distraction
    $executor.invoke($deskOfTeacher, "StartGame",
                     [$deskOfGeorge,  "TheSubtractionGame", "Easy"],
                     [$deskOfGeorge,  $context]).then(function() {

      $tester.test("Starting game for George:", function() {
        $gamesOnDeskOfGeorge = $deskOfGeorge.getGamesRunning();
        $tester.expect($gamesOnDeskOfGeorge.getKeys()).toContain("TheSubtractionGame");
      }).run();
    });
  // Expect for the tests to be executed properly within 5000 ms.
  }, 5000).onAnyFailure(function() {
    $tester.fail("Test failed! Could not start game to address George's distraction.");
  });
};
```

Figure 4. Portion of the automatically generated ClassScript code to test SLE reaction to a student's lapse of attention

Upon programming of the classroom's behavior and the testing methodology, Mrs. Smith launches the validation process. She uses the TMDS component of the AmITest framework to create an instance of that test and execute it right away in order to check whether or not the SLE behaves as expected.

VI. 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 the 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 operated as expected. For instance, 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

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-471-8

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.

To support such tests, we have implemented a sophisticated monitoring mechanism through which we ensure that value checking (i.e., Expectations) is performed only after the necessary handling actions have completed (i.e., Promises).

VII. CONCLUSIONS AND FUTURE WORK

This paper has described a testing suite for Smart Learning Environments in order to check the validity of operations programmed by end users in a Smart Learning Environment. The suite aims primarily at 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., Blockly), following the iterative approach of the User-Centered Design (UCD) process [29], both educators 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. Whereas, 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 educators in order to examine and improve the usability of the AmITest editing facilities.

Finally, as regards our future plans for the overall framework, we have already laid the foundations to extend its application to support testing, in a scalable and effective way, in other domains beyond its initial target domain (i.e., SLEs), such as Smart Homes, Technologically-enhanced Cultural Monuments, Smart Cities, etc.

REFERENCES

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

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

[3] G. M. Novak, Just-in-time Teaching: Blending Active Learning with Web Technology. Upper Saddle River, NJ: Prentice Hall, 1999.

[4] M. Chang, and Y. Li. Smart Learning Environments. Springer, 2014.

[5] A. Leonidis, M. Antona, and C. Stephanidis, "Enabling Programmability of Smart Learning Environments by Teachers." Distributed, Ambient, and Pervasive Interactions Lecture Notes in Computer Science, 2015, pp. 62-73.

[6] D. Crockford, JavaScript: The Good Parts. Beijing: O'Reilly, 2008.

[7] P. Ragonha, Jasmine JavaScript Testing: Leverage the Power of Unit Testing to Create Bigger and Better JavaScript Applications. Birmingham: Packt, 2013.

[8] D. Sheiko, Instant Testing with Qunit. S.l.: Packt Publishing Limited, 2013.

[9] M. Maleki, R. Woodbury, R. Goldstein, S. Breslav, and A. Khan. "Designing DEVS Visual Interfaces for End-user Programmers." Simulation 91, no. 8 (2015), pp. 715-734.

[10] M. Resnick et al, "Scratch: Programming for All." Communications of the ACM Commun. ACM 52, no. 11 (2009), pp. 60-67.

[11] 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.

[12] 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.

[13] 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.

[14] B. Waldie, Automator for Mac OS X 10.6 Snow Leopard. Berkeley, CA: Peachpit Press, 2010.

[15] R. Ierusalimschy, Programming in Lua. Rio De Janeiro: Lua.org, 2006.

[16] W. Goldstone, Unity 3.x Game Development Essentials: Game Development with C♯ and Javascript. Birmingham, UK: Packt Publishing, 2011.

[17] 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.

[18] E. Mangina, J. Carbo, and J. M. Molina, Agent-based Ubiquitous Computing. Paris, France: Atlantis Press, 2009.

[19] 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.

[20] A. Williams, C++ Concurrency in Action: Practical Multithreading. 1st ed. Manning Publications, 2012.

[21] D. Parker, JavaScript with Promises. O'Reilly Media, 2015.

[22] "Blockly | Google Developers." Google Developers. Accessed January 25, 2016. https://developers.google.com/blockly/.

[23] A. Leonidis et al, "A glimpse into the ambient classroom." Bulletin of the IEEE Technical Committee on Learning Technology 14.4, 2012, 3.

[24] M. Antona et al, Ambient Intelligence in the classroom: an augmented school desk. In the Proceedings of the 2010 AHFE International Conference (3rd International Conference on Applied Human Factors and Ergonomics), Miami, Florida, USA, 17-20 July 2010. CRC Press [CD-ROM].

[25] C. Savvaki et al, "Designing a Technology–Augmented School Desk for the Future Classroom." HCI International 2013-Posters' Extended Abstracts. Springer Berlin Heidelberg, 2013, pp. 681-685.

[26] G. Galanakis, X. Zabulis, P. Koutlemanis, S. Paparoulis, and V. Kouroumalis, "Tracking persons using a network of RGBD cameras." In Proceedings of the 7th International Conference on PErvasive Technologies Related to Assistive Environments, ACM, 2014, 63.

[27] M. Korozi et al. "Ambient educational mini-games." Proceedings of the International Working Conference on Advanced Visual Interfaces. ACM, 2012.

[28] G. Mathioudakis et al. "Ami-ria: real-time teacher assistance tool for an ambient intelligence classroom." Proceedings of the Fifth International Conference on Mobile, Hybrid, and On-Line Lerning (eLmL 2013). 2013.

[29] D. A. Norman and S. W. Draper. "User centered system design." Hillsdale, NJ, 1986.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-471-8