A Model-based Approach to Ongoing Product Evaluation

Shah Rukh Humayoun  
Dipartimento di Informatica e Sistemistica "A. Ruberti"  
SAPIENZA - Università di Roma  
humayoun@dis.uniroma1.it

Yael Dubinsky  
IBM Research - Haifa  
Mount Carmel  
Haifa 31905  
dubinsky@il.ibm.com

Tiziana Catarci  
Dipartimento di Informatica e Sistemistica "A. Ruberti"  
SAPIENZA - Università di Roma  
catarci@dis.uniroma1.it

Eli Nazarov, Assaf Israel  
Computer Science Department  
Technion – Israel Institute of Technology, Haifa 32000  
eli.nazarov@gmail.com  
assafi@cs.technion.ac.il

ABSTRACT
Evaluating interface usability and system functionality is time-consuming and effort-intensive. The short time-span involved in development iterations, such as those in agile development, makes it challenging for software teams to perform ongoing interface usability evaluation and system functionality testing. We propose a way to perform product ongoing evaluation, thus enabling software teams to perform interface usability evaluation alongside automated system functionality testing. We use formal task models, created in our defined TaMoGolog task modeling language, to conduct product evaluation experiments through TaMUlator. TaMUlator is a tool we developed for use at the Integrated Development Environment (IDE) level. Our case study revealed that software teams can easily engage in system evaluation by using TaMUlator on an iterative basis.

Categories and Subject Descriptors
D.2.6 [Programming Environments]: Integrated environment, H.5.2 [User Interfaces]: Evaluation/methodology, K.6.3 [Software Management]: Software development, software process.

General Terms
Management, Measurement, Design, Languages.

Keywords
Product evaluation, usability evaluation, automated testing, task models, TaMoGolog, TaMUlator.

1. INTRODUCTION
Usability evaluation aims at involving users in the evaluation process of a specific product to find usability flaws and errors and refine the product according to their feedback. Such users usually include product end-users, UI experts, and system analysts. Usability evaluation is performed using existing rigorous approaches and techniques to define and run experiments, collect and analyze results, and make decisions regarding which feedback to adopt and to what extent [6]. In many cases, these usability evaluation techniques are performed manually [14]. Automating evaluation approaches and techniques and applying them throughout the development process provides several benefits. These include reduced development costs and time, improved error tracing, better feedback, and increased coverage of the evaluated features [14].

Contemporary approaches recommend starting the testing of system functionalities earlier in the development phase, although this is traditionally performed at the end of the development lifecycle [1, 31, 32]. When adopting this practice, i.e., developing and maintaining system tests throughout the development lifecycle, we can see that usability evaluation and functionality naturally merge and contribute to each other. This motivated us to see these two activities as part of the product ongoing evaluation that evolves alongside the actual product development.

We propose an approach that enables software development teams to effectively and simultaneously conduct both interface usability evaluation and automated system functionality testing, thus saving the time required to perform this testing separately and encouraging its performance alongside development activities. We use the term product evaluation for the evaluation of both interface usability and testing of system functionality. We run user-based product evaluation experiments to find interface usability issues and to serve as a kind of acceptance test for the developed features and functionalities.

Moreover, we propose the integration and automation of the evaluation process into the Integrated Development Environment (IDE). This equips the development teams with the mechanisms to monitor and control a continuous evaluation process tightly coupled with the development process. This process will keep the time constraints of short-term development iterations well in control.

We already suggested this IDE-level integration and automation in [7, 11], using the UMan tool presented in [10]. While the UMan tool does provide an effective framework to automate the evaluation process, it cannot formally model user evaluation scenarios. This results in experiments not being formalized to enable automatic analysis of usability issues and system functionality problems. In this work, we focus on how we can use formally defined user and system activities and behavior during modeling user evaluation scenarios for the automatic analysis of the experiment, for recording the data to enable interface usability evaluation and to test the system functionality.

To enable effective product evaluation in software development, we use formal models. The models were created in our defined TaMoGolog task modeling language. We merge the interface usability evaluation and system functionality testing in the same evaluation experiment by specifying tasks in the application code at multi-abstraction levels in which a task can be a simple command, a user input, a system function/method, or any code part. We also provide the modeled user evaluation scenario with the task execution paths, the precondition axioms to the task (where these axioms can contain every kinds of constraints), and the postcondition effects to variables (fluents) due to successful task execution. To realize this approach, we developed a tool
called TaMULator. The TaMULator tool enables the proposed approach’s end-to-end evaluation lifecycle, including specifying tasks in the application code, defining the user evaluation experiments using TaMoGolog-based task models, and running the experiments. This allows us to record user and system activities and behavior as per the defined mode and then automatically analyze the usability issues and system functionality problems. We provided an initial introduction to the tool in [13]. In this work, we emphasize the different aspects of our product evaluation approach and provide details of a case study in which six development teams successfully used the approach alongside the TaMULator tool to evaluate products they developed. Based on the usability issues and functionalities drawbacks they found, the teams drew conclusions to derive relevant development tasks for the forthcoming iterations for further improvements to their product.

The remainder of this paper is structured as follows. In Section 2, we briefly introduce the TaMoGolog task modeling language, which we use to formally model user evaluation scenarios. In Section 3, we explain the different aspects of our proposed Task-based Product Evaluation (TaPE) approach, including the TaMULator tool. In Section 4, we describe our case study. In Section 5, we list other tools and approaches for usability evaluation and examine how they differ from our approach. We conclude and shed light on future directions in Section 6.

2. TaMoGolog: A FORMAL TASK MODELING LANGUAGE

The TaMoGolog (Task Modeling Golog) [12] task modeling language was built on top of the foundations of the Golog family [4, 5, 17] of high-level programming languages. TaMoGolog provides well-defined syntax and semantics, enables precondition axioms of task, states postcondition effects to variables due to successful task execution, provides a rich set of operators, and gives the facility to express domain knowledge in task models. TaMoGolog distinguishes tasks using three main categories: unit tasks, denoted as \( \alpha \), which are performed in an atomic manner; waiting tasks, denoted as \( \alpha_w \), which wait either for a particular event to happen or for some set of conditions to be fulfilled; and composite tasks, denoted as \( \Gamma \), which handle the structural behavior of the task model.

TaMoGolog provides a rich set of operators, mostly obtained from the Golog-family languages. Operator \( \phi \) represents a waiting or testing condition; operator \( [\Gamma_1]; [\Gamma_2] \) represents sequence in which task \( \Gamma_2 \) starts after task \( \Gamma_1 \) is finished; and operators \([\Gamma_1][\Gamma_2]\) and \([\text{agt}\ \Gamma_1][\Gamma_2]\) represent internal and external nondeterministic choice, respectively. In this case, \text{agt} represents some external entity (application/system or human user) that decides the nondeterministic branch. Operator \([\text{if } \phi \text{ then } \Gamma_1 \text{ else } \Gamma_2]\) is a normal if-then-else choice in which \( \phi \) is a conjunction of conditions. Operators \([\pi. \Gamma(x)]\) and \([\text{agt}\ \pi. \Gamma(x)]\) are nondeterministic choice of argument in which the system or the external entity chooses the variable binding for the task and then the task is executed accordingly. Operators \([\Gamma]^*\) and \([\text{agt}\ \Gamma]^*\) are nondeterministic iterations (internally and externally). Operator \([\text{while } \phi \text{ do } \Gamma]\) is the usual while-do iteration. Operator \([\Gamma_1][\Gamma_2]\) represents the interleaving concurrency of tasks while operator \([\Gamma_1]!\alpha[\Gamma_2]\) represents the priority concurrency in which task \( \Gamma_2 \) continues only if task \( \Gamma_1 \) is finished or is in the blocking state. Operator \([\Gamma]^!\) is for concurrent iteration while operator \([\text{agt}\ \Gamma_1]^!(\alpha[\Gamma_2])\) represents that the external entity \text{agt} decides the priority concurrency at run time. Operator \([\text{agt}\ \Gamma]^!\) represents the concurrent iteration decided by external entity while operator \( <\phi \rightarrow \Gamma >\) represents interrupt where when the condition \( \phi \) becomes true and the interrupt has control and the task \( \Gamma \) is started. Operator \([\Gamma_1]!\alpha[\Gamma_2]\) represent failure handling, when \( \Gamma_2 \) executes if \( \Gamma_1 \) fails to finish.

The TaMoGolog comprises the following selected predicates useful for writing task models with our approach:
- \text{UnitTask}(\alpha): \( \alpha \) is a unit task
- \text{CompositeTask}(\Gamma): \( \Gamma \) is a composite task
- \text{Precondition}(\alpha): \( \Gamma \) is a composite task
- \text{Postcondition}(\alpha, \nu, \Omega_{\alpha})(\nu): formula \( \Phi_{\alpha}(\nu) \) defines the set of axioms that must be true at the time of execution of a unit task \( \alpha \)
- \text{Fluent}(f, \nu): \( f \) is a functional or relational fluent (variable)
- \text{InitialState}(m) = \( \Omega_m \): describes the values of variables at the beginning of a task model \( m \)
- \text{TaskModel}(m): \( m \) is the task model name
- \text{Agent}(\text{agt}): \text{agt} represents some external entity (e.g., human or application) participating in the task model
- \text{Responsible}(\text{agt}, \alpha): \alpha \) is responsible for executing task \( \text{agt} \)
- \text{Goal}(g, \nu): \text{goal} \( g \) is defined by formula \( \Lambda_{g, \nu} \) on fluent \( \nu \)

TaMoGolog semantics is also based on the Golog-family languages, in which a unit task executes if all precondition axioms are true; then the effects are shown on related variables (fluents) after execution. Unit tasks are performed in an atomic manner while composite tasks are performed in step-by-step executions. Each task model defines one or more paths (depending on nondeterminism) to achieve a goal. If a task model achieves a goal through any of the possible paths, then it succeeds; otherwise, it fails.

A Task Model Example: Figure 1 shows a simple task model for managing a bank account scenario in TaMoGolog, with limited details. The task model structure defines that a bank customer can check her account after logging in to the system and can then view
her account and deposit or withdraw money as often as she wants until she logs out of the system.

3. A TASK-BASED PRODUCT EVALUATION APPROACH

We propose an approach called Task-based Product Evaluation (TaPE), by which we evaluate both interface usability and testing of system functionality. We developed a tool called TaMUlator to automate the proposed lifecycle from within the development environment. We use user-based evaluation experiments to find usability issues and serve as a kind of acceptance test for the developed system features and functionalities. These experiments are preferably performed by end users, although they could also be done by having experts perform tasks on the targeted application to achieve the desired goal(s).

Our TaPE approach merges the interface usability evaluation and system functionality testing in the same evaluation experiment by specifying tasks in the application code at multi-abstraction levels and by formally modeling the user evaluation scenarios through TaMoGolog. The proposed approach uses tasks (either specified in the application code or defined in the task model), the precondition axioms to the task (in which these axioms can contain every kind of constraints), and the postcondition effects to variables due to successful task execution for recording user and system activities and behavior during experiment execution and then for automatically analyzing the usability issues and functionality problems. In the following sections, we address different aspects of our proposed TaPE approach to understand how we can enable product ongoing evaluation through evaluation experiments using TaMoGolog-based task models. We then describe how our TaMUlator tool automates the evaluation process from within the development environment.

3.1 Specifying Tasks in the Application

We propose specifying tasks in the application at the code level. These tasks are then used for three purposes: 1) in task models of user-evaluating scenarios, 2) for collecting data during experiment execution when they are enabled by users or system, and 3) during automated analysis to find out usability issues and functionality problems. Alongside specifying tasks in the application code, we also propose tagging the interested set of variables (fluents) that can be either part of precondition axioms or postcondition effects of these specified tasks. The tags can then be used during modeling evaluation scenarios.

This approach of code-level specifying tasks gives the freedom to define tasks from user input, such as by clicking a button or selecting an option, to some system function, such as payment transaction function, to other functionalities, such as interacting with some external service. Moreover, it enables us to specify tasks at multi-abstraction levels, from a single command to a complete function. This is very useful for testing system functionalities as it enables us to define any part of the application as a task. Hence, through related precondition axioms and postcondition effects in the created task models, we can easily check whether system functions are working as they are supposed to be. Through this process, we can perform different levels of functionality testing, such as white-box or black-box [30], as it enables specifying a complete function as well as inside parts of the function as separate tasks. Interestingly, interaction parts and inside commands of user interface can also be defined as separate tasks. In a nutshell, this approach of specifying tasks in the application code provides a foundation to test system functionality alongside evaluating interface usability.

3.2 Modeling User Evaluation Scenarios

Our approach uses user-based evaluation experiments, performed by end users, to conduct product evaluation. A user evaluation scenario in these experiments provides the scenario that end users should follow during evaluation, consisting of a series of tasks on the targeted application, to achieve the goal(s). The purpose is to collect the user and system actions and behaviors during the experiment execution and then to analyze the logged data against the created formal models of these scenarios to highlight usability issues, such as user performance or system error handling, and to reveal functionality problems, such as a function giving wrong output. This analysis is done by applying appropriate usability metrics and testing measurements.

We use TaMoGolog to model user evaluation scenarios. Through our approach of specifying tasks in the application code, we are able to model these scenarios more dynamically and accurately. Moreover, we use our approach of constructing task models from multi-view perspectives as defined in [12], which helps to collect data during experiment execution from many perspectives for the same scenario. This is useful in finding out the usability issues from different abstraction levels, hence giving a clearer picture of the reasons behind the observed usability issues. It also greatly helps in testing system functionality, as it enables conducting different kinds of testing, such as white-box or black-box. For example, specifying a complete function as a task and with the help of postcondition axioms in the created task model, we are able to discover whether the function behaves correctly during execution. On the other hand, we can specify commands inside the function as tasks. Then, through analyzing task-execution paths of the users during experiment execution, we are able to test the execution paths of the scenario/function to check different execution coverage in addition to understanding users and system behavior.

3.3 The Data Recording Process and the Data Analysis Criteria

The proposed TaPE approach uses tasks and variables specified in the application code and the created task models of user evaluation scenarios, first for recording users’ and system activities and behavior during experiment execution, and then for the automatic analysis of usability issues and the testing of system functionality.

We record data when a task, specified in the application or modeled in any of the scenario task models, is enabled during experiment execution. This logging data includes: variables’ values related to precondition axioms; the necessary information of a task such as starting time, previous task, completion time, etc; and the postcondition related variables’ values before the task is started and when it is completed. We also record the inputs given or nondeterministic decisions taken by the system or by an external entity (some external application/system or human user). Other recording data include values of all those variables that come in any of the attached task models at the beginning and the end of the evaluation experiment (for system initial state and ending state values), and other domain-relevant information.

For the automatic analysis of usability issues and the testing of system functionality, we use TaMoGolog-based task models of the targeted user evaluation scenarios and the recorded data.

- Our approach incorporates many measurements suggested by Whiteside, Bennett, and Holtzblatt [28] for evaluating usability. These include: time to complete a task, percent of task completed, percent of task completed per unit time, ratio
of success to failure, etc.; taken directly from the experiment’s recorded data by applying the appropriate statistical techniques. Many of these measurements were also implemented in our previous work [7, 10] for the automation of other usability evaluation techniques.

- We categorize tasks in the evaluation experiment logged data into four categories for the analysis purpose: completed-and-successful, completed-and-unsuccessful, failed, and avoided. Completed-and-successful refers to a task that finished execution properly in the experiment and that its recorded attached postcondition variables’ values satisfy the attached postcondition effect axioms in the task model. Completed-and-unsuccessful refers to a task that finished execution properly in the experiment but that its recorded attached postcondition variables’ values do not completely satisfy the attached postcondition effect axioms in the scenario’s task model. The postcondition effects in both of these cases are used for testing whether the tasks are working as they should, so in the second case, they should show a failure of system functionality testing. Failed refers to a task that failed to successfully finish in the experiment or that its user abandoned it midway. Avoided refers to a task that was in the execution-path, but that its evaluating user chose an alternative path or chose the defined execution-path but decided not to perform it. In all cases, we use tasks’ recorded data, precondition axioms, postcondition effects, and the related variables’ values collected during experiment execution to analyze the reasons behind usability issues and system functionality problems.

- For tasks specified for system functions, we check the attached postcondition variables’ values to determine if these system functions/actions performed as expected.

- We also analyze the users’ selection of choices/tasks and the attached task models to evaluate the users’ execution-path selection behavior.

- Using the exogenous actions’ data (actions performed by external systems/applications), we investigate the logged data to see the effects of these actions on system states and on other tasks.

In addition to the above, our proposed TaPE approach also uses other information such as user groups, their particular preferences, and so forth, which are provided in the created task models for different data analysis purposes.

3.4 TaMUlator: A Tool for Automating TaPE Lifecycle

We developed TaMUlator, a Java-based tool that works at the IDE level to automate the proposed TaPE lifecycle. This realization at the IDE level helps the development teams to manage product ongoing evaluation from within their development environments, as already suggested in [7, 10], thus giving them deep and continuous control over the evaluation process while developing. Currently, the TaMUlator environment supports product evaluation of Java-based applications, as the current version only provides a Java-based library of APIs [13]. This library helps development teams to define and execute user-based product evaluation experiments throughout the product development. In the next section, we explain the task-specifying process through TaMUlator and how the TaPE lifecycle manifests itself in the TaMUlator. We illustrate this process with a simple example.

3.4.1 TaMUlator Tagging Process

First, to specify tasks and related variables in the application code, called the tagging process in TaMUlator, developers need to “wire” their programs in “interesting” locations (for tasks and variables) throughout their code. The TaMUlator tool leaves it to the software team to define those points of interest (e.g., methods invocations, object state changes, or certain events). This tagging process can be done in one of two ways. Through the task tagging (TTag) API provided by TaMUlator, developers can tag tasks and related variables by simply stating their names. Alternatively, they can use Java AspectJ facility, which is a Java extension that supports the aspect constructs separating cross-cutting concerns from an object-oriented system and providing a mechanism to merge these aspects into the underlying system. Through this, developers can define multiple pointcuts and advices that inject small snippets to let TaMUlator know that an event has occurred. From a development perspective, this is especially handy, since the AspectJ code is external to the program code. These aspects can be weaved in and out of the code in a dynamic fashion, thus allowing the evaluation process to be done during any phase of the development process and even after the product deployment.

3.4.2 How TaMUlator Automates TaPE Lifecycle

To illustrate how TaMUlator automates TaPE lifecycle, we present a simple user evaluation scenario. In our scenario, the development team is presented with a task of evaluating the login page of a website. Such a page may involve various backend methods (e.g., the login method), variables, the user credentials, and the front-end interaction, etc. Some of these may be points of interest the development team would want to follow. Figure 2 shows a simple AspectJ snippet for telling TaMUlator to record its data when the login page is activated.

![Figure 2. A typical AspectJ hook for the login web page](image)

```java
pointcut login(LoginManager framework) : 
  call(public UserProfile LoginManager.login(..))  
   && target(framework); ... TaMUlator.activityOccured("login"); 
} catch (TaskModuleException e) { 
  System.err.println(e.getMessage()); 
 })
```

![Figure 3. A simple task model for the login scenario](image)

```java
Precondition-axiom(username) = username != "";
Label login-page { 
  login; 
  if user_credentials_approved? then 
    display_welcome_page; 
  else 
    display_error_page; 
  end 
}
```

The development team creates task model, which reflects the scenario of evaluating login page, using TaMoGolog syntax. Figure 3 shows a simple task model for this scenario (Label name {...}) is syntax for composite task in TaMUlator. The process of creating task models in TaMUlator is very simple. It entails writing a TaMoGolog-based script, using the available set of

1 http://www.eclipse.org/aspectj/
tagged tasks and variables (specified in tagging process) and defining the task structure with temporal relations in tasks, precondition axioms, and postcondition effects, and letting TaMUlator compile it. The TaMUlator APIs [13] make it easy to aggregate compiled task models into experiments, to enable them for tracking, and finally to analyze the result and export them into a convenient format for subsequent analysis.

When the evaluation experiment is executed and as the evaluating user tries to login, method invocation and variable changes are caught by AspectJ snippets. The snippets then send alter of changes in the tagged task and variables to the TaMUlator. The TaMUlator records these actions and other data, and later analyzes the log against the associated task model.

3.4.3 The TaMUlator Automatic Analyzer
TaMUlator uses automatic analysis of the recorded data to reveal usability issues and functionality problems. A built-in automatic Analyzer evaluates the experiment results after comparing them with associated task models of evaluation scenarios. The recorded data can also be exported into a CSV format for manual analysis. At any time, the software team/evaluator can issue an analysis of the recorded data against any task model or experiment.

The current version of the TaMUlator Analyzer [13] provides partial support of the TaPE analysis criteria (described in the previous section). The automatic Analyzer determines if the recorded data are consistent with the task models, by checking that the recorded tasks appear in the same order as in the task structure, ensuring that the precondition axioms were met, and that the postcondition variables possess the desired values. If any of these conditions are not met, the scenario for the task model is considered not properly fulfilled, as compared to what the designer/evaluator wanted. This helps the development team/evaluator to find usability issues and functionality problems in the evaluated application and is especially useful for locating the points at which users tend to make mistakes or at which the system is not functionally proper. Thus the development team can take care of these places in forthcoming development iterations.

4. A CASE STUDY
In this section, we present a case study in which six development teams (composed of 6-7 students each) used our TaPE approach along with the TaMUlator tool to evaluate the software project they developed2. We gathered data using the course materials, e.g., product requirements and exercises; project artifacts, e.g., iteration backlogs, task models, and experiments’ results3; and written materials by team members, e.g., communications in web forums, feedback of role holders, and the final course retrospective.

The development approach used is based on agile development [1] and is presented using three main perspectives: human/social (H), organizational (O), and technical (T). More information about the HOT framework that provides case analysis using these three perspectives can be found in [9].

The main ideas fostered as part of the human/social perspective are teamwork, collaboration, and reflection. Teams meet every week for a four-hour compulsory meeting at which they communicate regarding the product development and the process management. Team members periodically reflect on their activities. As part of the organizational perspective, the project was defined for two releases. Each release comprised two iterations of three weeks (i.e., four development iterations). Roles were ascribed to each of the team members as part of his/her team management, e.g., in-charge of unit testing, tracking, or designing (see more details regarding the role scheme in [8]). Role holders presented measures for the relevant responsibilities. As part of the technical perspective, the following practices were used: automated testing, continuous integration, and refactoring to ensure simple design. The role scheme supported these practices by emphasizing the appropriate practices for specific iterations and changing the role scheme accordingly in other iterations. For instance, the person in charge of continuous integration worked mainly at the first iteration to provide the infrastructure and work procedure, while refactoring activities were the responsibility of the designer at the third iteration, and so on.

The combined interface usability evaluation and functionality testing (which we call product evaluation) of a product in development is yet another practice that was implemented as part of this project. Based on our experience with guiding the implementation of the agile approach [8, 9, 27], and the integration of user-centered design (UCD) techniques and usability engineering in the last four years in agile projects in industry and academia [7, 9, 11], we used the following main practices: 1) iterative design activities, including cycles of development that contain development tasks derived from the results of product evaluation; 2) role holders in the subject of product evaluation and using the TaPE approach and TaMUlator tool; and 3) measurements that were taken by the role holders as part of fulfilling their responsibilities.

4.1 Brain Fitness-Room
The student development teams’ project, named the Brain Fitness-Room, aimed to develop a system that supports maintaining and strengthening memory and brain capabilities as well as identifying any decline in those capabilities. The motivation for such a system included retaining and improving brain capabilities; detecting brain illness, specifically dementia syndrome [24]; slowing down the progress of known and unknown afflictions; and preventing brain illnesses. The system included a pool of games that fit in with its goals, thus providing, among other benefits, fun for the users and enabling the collection of data for future studies.

The main components were defined as follows:
- The application provides three types of user interfaces for three types of users: players, doctors, and administrators. Players can log in, play a session or a specific game, view history, or get advice.
- A session builder can be used to create a session for the player in which he/she plays and is advised.
- A game pool is provided. Games can run on the system and can be added separately. A standard is required so all games can run on all systems (belonging to the six teams). The games comprise two types:
  - Left hemisphere medium/long-term memory game, e.g., studying a list of random words for a few minutes then writing down all word recalled after half an hour.

---

2 The team members are 4th year CS-major students participating in the “annual project in software engineering” course of the Computer Science Department at Technion, IIT.

3 The project material of all six teams can be found at http://www.cs.technion.ac.il/ssdl/projects/projects.html
Right hemisphere short-term memory game, e.g., studying a random shape for one minute then drawing it from memory.

- An automated built-in system advisor can issue warnings and suggestions by analyzing the collected data.

History and statistics are stored in a database and can be viewed upon request.

In addition to the above-mentioned requirements, the subject of product evaluation using formal task models was presented to the teams. In the first development iteration, all teams had to develop a tool to enable writing basic task models in TaMoGolog. An end-to-end scenario was defined. This was used to develop evaluation experiments based on TaMoGolog task models using an editor, execute them while recording user and system activities and behavior, and analyze the results based on a comparison between the task model and the recorded data while performing the experiment.

4.2 Evaluation Using TaMULator

TaMULator tool was used in the third and fourth iterations to evaluate the system that was developed. In this section, we illustrate how working with TaMULator is integrated with the agile iterative manner by presenting an end-to-end example of using the tool.

Towards An Iteration: The development team wanted to check how users activate the system functionalities. Among other factors, they decided to check the time spent on the different system capabilities to better understand user behavior. A few items were added to the coming iteration backlog, including one work item that dealt with creating a new TaMoGolog-based task model and experiment to evaluate the time spent in different brain games.

During The Iteration: Figure 4 shows the TaMoGolog-based task model that was defined by the development team as part of this work item. Analyzing the evaluation through time spent, when the precondition of five active sessions is met, they derived four measures: a) time elapsed between login and first play activation (as one indicator of the usability of the main menu), b) time spent on each game and whether the user completes the game, c) average session time, and d) average time in the system. The summary of results is shown in Table 1 and Figure 5 and was presented as part of the iteration summary.

Plan the Next Iteration: Based on these results, the team reached several conclusions and suggested several development tasks accordingly. In one case, the average time from login until start of game session (63.77 seconds) was relatively high, since all that was required from a new user (player) was to select the default session and start playing. The team suggested changing the user interface in a way in which it will be clear not to click the Play button before selecting a session. This included a new arrangement of the Session and Play buttons and a change in the usage of the Session check box in cases in which the user needs the default session. In a second case, the average play time of the Silly Color game (24.8 seconds) was too short. The team suggested adding a higher difficulty level to increase game effectiveness.

Release Retrospective: The release retrospective includes impressions and thoughts that relate to the entire experience of working in an agile environment in which product evaluation is implemented. In this retrospective session, team members were asked to grade their satisfaction from 1 to 5 (very satisfied) with respect to the project topic, methodology (agile, time management, and early detection of problems, emphasis on performing testing and usability together), tools that were used (Trac and Moodle), and the services in the physical lab where they worked. Most team members (32) answered, and the average grade for the methodology was high (4.09: 4.36 for project topic, 3.98/3.28 for tools respectively, for tools, and 2.66 for lab services). Specifically regarding the roles concerned with product evaluation, team members referred to the importance of learning and dealing with usability and testing while developing. Some of their comments on this matter include: “It is important to get feedback from the users...”, “It does not matter how good the product is, [people] will use it only if it is simple and user friendly”, “A lot of things that seem clear to developers are not clear to the end users”, and “The role of being in charge of the evaluation experiment was an important role with which we specified the usage of our system by the user”.

![Figure 4. Evaluate time spent in different brain-games](image)

**Table 1. Time measures for each game**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Average Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login until logout for regular user</td>
<td>412.55</td>
</tr>
<tr>
<td>Login until logout for doctor</td>
<td>56.88</td>
</tr>
<tr>
<td>Login until start of game session</td>
<td>63.77</td>
</tr>
<tr>
<td>Average game time for all games</td>
<td>74.38</td>
</tr>
<tr>
<td>Bird game average play time</td>
<td>43.66</td>
</tr>
<tr>
<td>Piano Kombat average play time</td>
<td>121.22</td>
</tr>
<tr>
<td>Pirate Memory average play time</td>
<td>107.83</td>
</tr>
<tr>
<td>Silly Color game average play time</td>
<td>24.8</td>
</tr>
</tbody>
</table>

![Figure 5. Time spent game playing](image)
5. RELATED WORK AND DISCUSSION

Automating the evaluation process (either a usability evaluation or functionalities testing) is not a new phenomenon. A decade ago, Ivory and Hearst [14] conducted a detailed survey on the state-of-the-art in the automation of usability evaluation techniques on different levels, using the automation taxonomy (none, capture, analysis, and critique) suggested by Balbo [2]. They found that only 33% of the 132 selected techniques are supported by automated tools. Their survey concluded that usability evaluation methods automation is greatly underexplored. Even after a decade, the situation has not dramatically changed, especially in the usability evaluation area, as many methods in this area are still performed manually. In this section, we first describe other approaches and tools that also perform evaluation using the task-based approach. Second, we examine the differences between our approach and past approaches and illustrate why our proposed approach is well-suited for iterative and incremental development processes.

ConcurTaskTrees (CCT) [19] is the most widely used graphical-representative task modeling technique in model-based usability evaluation. Its operators are based on LOTOS [15] formal notations and enhanced by Sinnig et al. [26] by adding stop, nondeterministic choice, deterministic choice, and instance iteration operators. AMBOSS [23] is a task modeling approach and environment that focuses on the circumstances of safety critical systems. To do this, it keeps additional elements in task models such as spatial information, timing, and communication.

The USINE usability evaluation tool, developed at CNUCE-NCE (Pisa) [16], takes the task models (generated in XML format in CCT) and the logs generated by users (through the log recording tool Replay). The designer creates a log-task table for mapping physical actions performed by users to the basic tasks of the task model. USINE then automatically creates the precondition table, emphasizing the possibility of doing one task before others. The tool takes the users’ log files during experiment execution and gives the results after analyzing the user data with the task model, log-task table, and precondition table. RemUSINE [18] is an extension of USINE and provides support of remote usability evaluation. A recent enhancement is the MultiDevice RemUSINE tool [20] for mobile applications. AWUSA [3] targets already-built websites and assigns the evaluator/architect with the task of discovering task models of the targeted website.

ReModEI [3] has a client-server architecture for remote usability evaluation. The designer creates a dialog graph (for the specific device, e.g., PDA or notebook) to provide a corresponding concrete user interface to the client. On the client side, the system captures users’ interactions with the proposed user interface and returns them to the server, which analyzes them with the created task models. DiaTask [22] is used to develop a dialog graph to represent the navigation structure of the application, based on the already-specified task model. This approach was further enhanced in [21] for usability evaluation of smart environments.

Palanque et al. [29] provide a model-based approach for supporting usability evaluation through formal analysis of models, simulations, and the log data. Furthermore, the approach is supported by a model-based CASE tool. In [25], Bernhaupt et al. discuss the usability evaluation of multi-modal interfaces while using eye-tracking technique.

When we examine past task-based evaluation approaches, we discover that they concentrate on evaluating only the usability aspects of the product. This leads to one of the reasons that development teams tend to neglect these aspects—due to the brief nature of development iterations, teams usually do not have enough time to perform usability evaluations and system functionality testing separately. Because our proposed approach performs these functions together in the same evaluation experiment, it saves time and resources. Specifying tasks, modeling user evaluation scenarios, and running evaluation experiments, as suggested by our approach, are also well-suited with current iterative development since they enable developers to get user feedback and refine products accordingly. Product evaluation based on task models can be considered as part of the automated test suite, which in general gives ongoing feedback regarding product evolution.

On the application level, our proposed approach differs from past approaches in many aspects and provides several benefits, particularly considering the current development environments. One main difference from previous tools is that our TaMULator tool works at the IDE level to manage and automate the complete evaluation lifecycle, and is thus well-suited to current development teams, such as agile teams, giving them full control to carry on evaluation activities alongside development activities.

One of the novelties of our approach as compared to past ones is the fact that tasks and variables (related to precondition axioms or postcondition effects) specifying process (called tagging process in TaMULator) takes place in the application code. Defining system functions/actions as tasks plays a vital role in performing system functionality testing during product evaluation.

Moreover, modeling evaluation scenarios through TaMoGolog in evaluation experiments provides several benefits, including the following:

- The use of precondition axioms of tasks in evaluation: In previous approaches, preconditions normally referred to the checking of temporal relations between tasks. In our approach, it is handled through TaMoGolog operators in composite tasks, and the precondition axioms are used to determine if other conditions have been met before performing the task. This helps analyze the reasons behind failures and other usability flaws related to a particular task.
- The use of postcondition axioms, which reflect the effects on variables due to task execution, during evaluation. This highlights whether the task executed perfectly (by the user or system) and performed the expected functionalities. This also helps to determine if unexpected changes on variables are due to implementation flaws, thereby providing the functionality testing of the targeted application.

6. CONCLUSIONS

In this paper, we presented an approach for performing both the usability evaluation and functionality testing in the same user-based evaluation experiments using TaMoGolog-based formal task models. We presented the TaMULator tool, which manages and automates the approach lifecycle at the IDE level. We also presented a case study in which six development teams in academia used our approach and tool to evaluate their developing product. Our TaPE approach, performing evaluation using formal task models, can also be considered as part of the automated test suite of development environments for product ongoing evaluation. Moreover, our approach is well-suited to the iterative manner of development as it enables developers to get user feedback and system functionality testing simultaneously and refine their products accordingly.

In the future, we intend to check the approach’s effectiveness in software projects of various scales, both in industry and in
academia. Secondly, we also intend to extend the TaMUlator Analyzer so as to provide automatic suggestions based on evaluation results for improvements in the product and the visual support for writing TaMoGolog-based task models.

7. REFERENCES