Crowd-Based Programming for Reactive Systems

David Harel, Idan Heimlich, Rami Marelly and Assaf Marron
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, Israel
Email: {david.harel, rami.marelly, assaf.marron}@weizmann.ac.il, himlich.idan@gmail.com

***Abstract*—End-user applications aimed at the public in general (mobile and web applications, games, etc.) are usually developed with feedback from only a tiny fraction of the millions of intended users, and are thus built under significant uncertainty. The developer cannot really tell a priori which features the users will like, which they will dislike, and which ones will help create the desired outcome, such as high usage or increased revenue. In these cases, providing adaptive capabilities can be the key factor in the application's success. Existing self-adaptive techniques can provide some of the needed capabilities, but they too must be planned, and leave the developers, and much of the development process, "out of the loop". We propose a development environment that allows the wisdom of the crowd to influence the very structure and flow of the program being created, by voting upon behavioral choices as they are observed in early versions of the working program. The approach still allows the developers to retain known desired behaviors, and to enforce constraints on crowd-driven changes. The developers can also react to ongoing crowd-programmed feedback throughout the entire lifetime of the application.**

Keywords-Software Engineering; Reactive Systems; MDE; Scenario-based Programming; Incremental Development;

I. Introduction And Related Work

A growing number of applications, especially those intended for mobile and web platforms, are aimed at millions or even billions of users. While most development methodologies call for a detailed elicitation process with customers and users, the developers of such applications do not have an effective way to communicate with the vast majority of the large number of potential users, as part of the software specification and design. They thus have to resort to other means for predicting their needs. Consider, for example, an adventure/quest game, in which users can also purchase helpful resources. Should a suggestion to buy a "power bottle" pop-up after losing a game or after winning? Or should it be constantly, but passively, displayed? Should a key to the treasure chest disappear if not used in time, or should it remain available forever? And when dealing with more practical applications, such as a chat-bot, an online-store or a collaboration platform, the considerations are, of course, different but the problem remains. Common practices often call for the developer to make such choices in advance or to manually collect and apply feedback from selected users of early versions.

Agile development techniques indeed focus on obtaining such feedback frequently from a select group of stakeholders. More elaborate approaches to requirements engineering call for systematic collection of user feedback on designated features. For example, in [1], a "social adaptation" methodology is proposed for the manual process of requirements engineering, where the various design choices are explicitly stated, the users vote on these, and computer-aided analysis of the responses helps accelerate the necessary modifications to the system. And, in [2] a "social sensing" approach is proposed, where users are tasked with actually collecting quality measures (such as user comfort) that are needed for making adaptation decisions, but which the system cannot collect on its own.

Another approach is to design the program so that it can behave in a variety of ways, and to apply an automated adaptation technique, such as reinforcement learning, to allow the program to dynamically modify itself based on the user's actual behavior. Such adaptivity may be global, i.e., common for large groups of users, or can be designed to fit individual-user needs.

The recent extensive survey [13] analyzes the state of the art regarding the role of crowdsourcing in software engineering. Based on this analysis the task of obtaining end-user feedback with respect to an already-developed application, seems to receive less attention.

Our research is motivated by the desire to have a development and runtime environment where (a) an unlimited number of users can constantly provide focused feedback on the behavior of released applications, in the form of suggested program changes; and (b) the rationale, functionality, and expected effects of the automatically-generated program changes are readily visible to the developers, who can then make informed decisions on when to allow, modify, or ignore the suggestions. Program behavior thus incorporates enhancements that are based on first-hand user feedback and comply with developer-specified goals and constraints. We term this kind of approach *crowd-based programming*.

To deal with the increasing complexity of software systems, and the uncertainty around the behavior of their environments, software engineers have turned to self-adaptivity, where the system can change its behavior and structure in response to changes in the environment, in the user requirements, and even in the system itself. As can be seen, e.g., in [14, 12, 4, 11], such adaptivity is often application-specific and requires specialized design, which

978-1-5386-4041-8/17 $31.00 © 2017 IEEE
DOI 10.1109/CSI-SE.2017.3

may or may not be in line with other architectural preferences. In particular, adaptive behavior that is based on machine learning using neural nets [15] requires very specific designs and flows. Involving users in the adaptation process was proposed in [6], where the user can influence the adaptation behavior both at run-time and in the long term by setting individual preferences. This aims to balance the required system adaptation and user control.

Such adaptive techniques are insufficient when the developer wishes to create a relatively small number of behaviors that will be essentially fixed for all or most users, and at the same time reflect design choices that accommodate the developers' intent as well as the goals and preferences of a large and varied audience.

In this paper, we show how a general application-independent design approach, namely, *event-based abstraction* and *scenario-based programming* (SBP), can, in addition to its other advantages, facilitate crowd-based programming approaches (as introduced above) by automatically translating systematically-collected end-user feedback into application code. We describe and evaluate our scenario-based crowd-programming approach and the initial web-based development and runtime environment we have built to support it.

II. Scenario-Based/Behavioral Programming

A basic principle of the scenario-based approach is to abstract and model the behavior of the system and the users as sequences of events. The infrastructure that generates input and output events (such as button click, text entry, and display of a screen object) is considered separate, and is only secondary in the system's design and analysis. Execution progresses in cycles, a.k.a. *supersteps*, where each user event is responded to by a sequence of system-generated events until there are no more system events that should be triggered. (In the present preliminary version of our work, each superstep consists of a user event followed by a single system event.)

Scenario-based programming (SBP), a.k.a. *behavioral programming* (BP) was first introduced in [3, 9], with the language of *live sequence charts* (LSC) and was later extended to procedural languages like Java, C++ and JavaScript (see, e.g., [10]). In SBP, system components are scenarios, each of which describes an aspect of system behavior, much in the way humans describe what a system should do, either to each other or in a requirements document. Such specifications can be natural and incremental, and they simplify formal analysis and (compositional) verification (see, e.g., [7]). Most relevantly to the present research, SBP is conducive to adaptivity using reinforcement learning [5] and automated non-intrusive program repair [8].

Formally, the SBP/BP scenarios are automata, or state machines. In each state, the scenario declares events that it *requests*, i.e., ones that it wants to be considered for triggering, and events that it *blocks*, i.e., forbids from occurring. A scenario also has a transition function, dictating, for each state and triggered event, the new state. All scenarios are executed in parallel, in lockstep. Following an event (user, environment or system) all affected automata transition to their new states, and resynchronize. An event that is requested by some scenario and is blocked by none is then triggered. If there is more than one such enabled event, selection is carried out according to some strategy, such as random, probabilistic, or one based on lookahead or on program synthesis. Affected scenarios then transition to new states and the process repeats. This execution method, termed *play-out*, yields integrated system behavior that complies with the scenario-based specification. SBP also enables *play-in*: an interactive process where the developer specifies scenarios by recording and editing interactions with the system being developed.

III. Crowd-based Behavioral Programming

In addition to event abstraction and scenario-based programming described above, our technique for crowd-based adaptivity builds on the following principles:

Underspecification and temporary behavior: While most system behavior would be fully specified, the developer can allow cases where multiple events are simultaneously enabled. User feedback then drives incremental refinement of the final behavior. Until it is fully specified, the system uses probabilistic event selection at points of underspecification, and does not get stuck there. A key assumption is that "bad" choices do not cause real damage and are quickly found and eliminated.

Programming with linear scenarios: Commonly, program modules in any programming approach, SBP/BP included, contain constructs for non-sequential flow, like loops or if-then-else statements. While this is obviously valuable, we maintain that creating complex specifications from sequential rules and from separate sequential exceptions thereto, is often also convenient, and aligns with the way people tend to describe behavior. A linear scenario is thus a sequence of events and states, reflecting one path of program execution. It can be coded manually or recorded in an interaction with the system. States contain declarations regarding event choices, as described below.

Probability-driven scenario execution: In standard SBP execution, in each state, one event that is requested and not blocked is selected according to some strategy. Here we propose that the declaration of requested and blocked events in each state, be replaced by a weighted vote, or score – positive or negative – for each system event (with some default for "don't care"). When scenarios synchronize, the then-current scores form a distribution, with high positive scores yielding high probabilities and negative ones yielding low probabilities. Below a certain score threshold, an event is considered forbidden altogether. The event selection decision is then random, according to the distribution. This facilitates showing each behavior to *some* users, who will

then be able to provide feedback. The probabilities can carry different meanings: the developer's confidence in various choices, the desire to force the system and its users to experiment with certain scenarios, or perhaps to make the final system behavior more varied and less predictable.

Ongoing user feedback: The system continuously grades its probabilistic event selection based on user feedback, which can be explicit, e.g., clicking *like* or *dislike* meta-buttons automatically superimposed on application screens, or implicit, where the application contains code that assigns event scores based on actual user behavior (like buying products, closing certain screens, etc.). The method thus captures first-hand users' reaction when they experience actual system choices, which may be better than having them comment on, say, system documentation or custom questionnaires.

Representing crowd feedback as linear scenarios: A user's feedback (whether explicit or implicit) is presented as a linear scenario, comprised of the events that led to a particular application state, followed by the scenario's vote/score for the various events that may be triggered next, with positive feedbacks increasing the scores and negative ones reducing them. The crowd-formed scenarios are self-standing, in that they modify overall application behavior without modifying existing scenarios. Also, they clearly convey to human users where and when they apply, and how they influence event selection at those points. This enables automated incorporation of such automatically-generated scenarios into system behavior, as well as informed manual adjustment by developers.

To illustrate these principles, consider a small online supermarket application, whose main screen is shown in Figure 1.

Figure 1. The product-list screen of the supermarket application, with Like and Dislike buttons

Clicking on any screen object (product image, action button) produces a corresponding user event. A scenario-based specification would include, e.g., the following scenarios (shown in pseudo code):

1. When(*user-clicks-on-product)*; Score(*show-prod.-info,*10*)*
2. When(*user-clicks-Buy)*;
   Score(*add-to-cart-&-show-prod. list,10)*
3. When(*user-clicks-Checkout*);
   Score(*checkout*,10,*show-ad*,1)
4. When(*user-clicks-Checkout*);
   Score(*checkout*,10,*show-ad*,1);
   When(*user-clicks-Close-Ad*);
   {Score(*checkout*,10,*show-ad*,-100);
   ExitUpon(*checkout*)}

Figure shows Scenario 4 above in its automaton form. The When() method lists one possible event and leads to the next state. The method Score(e_1,v_1,e_2,v_2,…) assigns to each system event e_i the corresponding score value v_i at that state. The method ExitUpon(e_1,e_2,…) can be used in a scenario to list events whose occurrence would cause the scenario to terminate (go back to its initial state).

Scenario 4, for example, allows for both the checkout screen and an advertisement to be shown. Users who prefer not to see the advertisement can click the *Dislike* button superimposed on it. When they do so, either a new scenario is created, which traces the session's event log and then assigns *show-ad* a low score, or, if such a scenario already exists, the score it assigns to the undesired event is reduced. If the developer is unsure as to whether a click on a product image should lead to *add-to-cart* or to *show-prod.-info*, he/she can assign positive scores to the respective events, and allow the users to vote. Similarly, the developer can use scenarios and/or specialized APIs to analyze when, or how often, users buy an advertised product, as opposed to skipping the ad, and accordingly create scenarios that increase or decrease the score of *show-ad* in certain states.

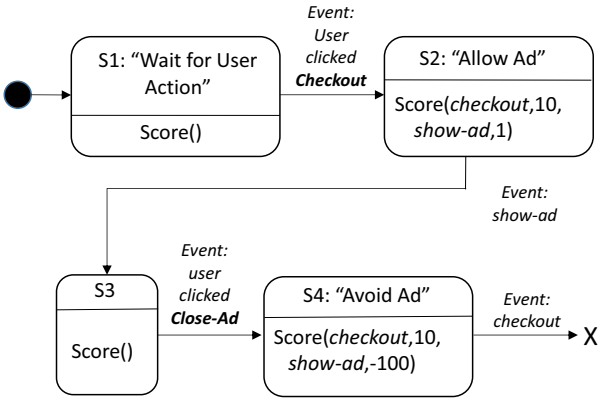

Figure 2. An automaton representation of a linear scenario: if in the initial state S1 the user clicks Checkout, State S2 assigns scores that cause either the checkout screen or an ad to be shown. In the case in which an ad is shown and the user skips it, state S4 forbids the showing of an ad until checkout screen is shown.

IV. THE CROWD PROGRAMMING INFRASTRUCTURE

Towards experimenting with crowd programming, we

have developed a new BP infrastructure with the following capabilities: (a) web-based; (b) automatic superimposition of feedback buttons on application screens; (c) automatic capture of feedback and creation and update of feedback scenarios with respective scores; (d) developer interface for specifying linear scenarios by manually simulating user actions and recording the resulting user and system events in line with play-in capabilities in LSC; (e) APIs for implicit feedback via tracking of user behavior and dynamically creating new scenarios; and (f) streamlined viewing and assignment of scores of all system events at all system states.

V. Evaluation and Discussion

In evaluating the approach, we sought to answer two main questions: (Q1) can the behavior for buying products in a web application be formed based on user feedback? (Q2) how easy will it be for a developer to understand the automatically-created application scenarios built this way?

In our initial experiment, we used the above online supermarket and another web-store application. In the initial code, we specified only most basic behaviors, and a wide variety of events was left enabled in each state. We also coded collection of implicit feedback that reflects developers' interest in maximizing user spending; i.e., the more the user buys the higher the score of system events in scenarios that lead to this behavior. The two applications were identical in their user interfaces, their events and the initial programmed behavior, and differed only in the offered products. However, we also told the users (participants) that "one application is a supermarket and the other is an online electronics store, like eBay", and asked them to set their expectations and behavior accordingly. The users were also told that the application's behavior may change based on their actual usage and their explicit feedback. In this initial evaluation, the users did not spend real money and did not receive actual products.

Over the course of five days, a group of ten participants experimented with the system. For the supermarket application, 67 *Like/Dislike* feedbacks were submitted, and 44 new scenarios were generated, and for the electronics store, 75 feedbacks were submitted and 43 new scenarios generated (we did not keep track of which user provided what feedback).

Regarding question Q1, when looking at the possibility of showing a special-deals ad, as expected, negative feedbacks were almost always received when the ad was shown after the user clicked on a product. However, after showing the shopping cart, showing the ad caused the user to buy more products, yielding indirect positive feedback. We also observed that the applications evolved differently: after the user clicked a product image, the evolved supermarket app added the product to the cart and returned to the main product list, while the evolved electronics store app proceeded to show product information and only then moved to checkout. This difference in the effect on similar applications shows that feedback has non-trivial value.

The second question, Q2, was not directly addressed by a specific test, but it is our subjective experience that despite the large number of generated scenarios, the developer could readily understand each of them and was able to create a mental model of the expected behavior of the evolving application. Moreover, the developer was able to simulate user behavior in specific flows, and check the scores assigned to the various system events to see which events were most or least likely to occur at a given point. As is common in SBP, the technique also allows one to see how the various scenarios affect the execution in each state.

We have thus described an application-independent, crowd-based technique for collecting user feedback during execution, and influencing future behavior accordingly. The modularity of the modifications enables a wide variety of implementation approaches. These include (a) choosing a fixed reactive behavior for all users based on a majority user vote; (b) retaining all behaviors with *some* probability alongside a feedback mechanism towards indefinite continuation of program evolution; and (c) providing users with application behavior that is uniquely geared specifically to their preferences.

Clearly, our current initial experience and evaluation results, while positive, are of limited scope. Issues that are left for future investigation include (a) using machine learning techniques like neural nets and enhanced event selection formulas to automatically aggregate crowd feedback into fewer and more general recommendation scenarios; (b) associating a set of feedbacks and preferences with the individual who communicated them, and using the information both in this user's initial exploration and during their regular use of the application; (c) supporting richer scenarios and supersteps, e.g., ones which respond to a user event with a sequence of system events (with flow-control) rather than by just one event; (d) supporting variable data in events and in the object model; (e) creating succinct guidelines for placement and granularity of feedback-collection points (*Like/Dislike* buttons) in an application, including dynamic changes to such feedback collection and possibly linking it to feature-models and software-product-line planning; and (f) allowing users to specify the exact target of their *Like/Dislike* vote, such as particular events in the execution log or some screen elements.

Furthermore, it will be interesting to extend the above into a broad, collaborative, concurrent, crowd-programming methodology, yielding short cycles of experimentation-feedback-modification-release. A key part of such a methodology will be the flexibility in the feedback means given to the end user. While some may be willing to suspend their work with the product to open a trouble ticket, with rationale, examples and attachments, others may wish to just enter a one-line comment to capture their reaction, while yet others may agree only to occasional, optional clicking on a *Like/Dislike* button, while their main work and train of thought remain focused. Eventually, such crowd-

based techniques may also be used to enhance methodologies and tools for unit testing, integration tests and system usability testing.

VI. Conclusion

Our crowd-based method for programming the behavior of reactive systems allows developers to subject some decisions to the wisdom of the crowd, while retaining sufficient control themselves. We believe that such an approach can become a key engineering practice in developing applications for large and varied audiences.

VII. Ackowledgement

This research was supported in part by a grant from the Israel Science Foundation (ISF).

References

[1] R. Ali, C. Solis, I. Omoronyia, M. Salehie, and B. Nuseibeh. Social adaptation: when software gives users a voice. 2012.

[2] R. Ali, C. Solis, M. Salehie, I. Omoronyia, B. Nuseibeh, and W. Maalej. Social sensing: when users become monitors. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 476–479. ACM, 2011.

[3] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 2001.

[4] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[5] N. Eitan and D. Harel. Adaptive behavioral programming. In *23rd IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI)*, pages 685–692, 2011.

[6] C. Evers, R. Kniewel, K. Geihs, and L. Schmidt. The user in the loop: Enabling user participation for self-adaptive applications. *Future Generation Computer Systems*, 34:110–123, 2014.

[7] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing and Proving the Correctness of Reactive Behavior. In *Proc. 13th Int. Conf. on Embedded Software (EMSOFT)*, pages 1–10, 2013.

[8] D. Harel, G. Katz, A. Marron, and G. Weiss. Non-Intrusive Repair of Reactive Programs. In *Proc. 17th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 3–12, 2012.

[9] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[10] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Communications of the ACM*, 2012.

[11] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[12] R. Laddaga. Active software. In *Self-Adaptive Software*, pages 11–26. Springer, 2000.

[13] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 2016.

[14] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Tran. on autonomous and adaptive sys. (TAAS)*, 4(2):14, 2009.

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