Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming

Michal Gordon
Dept. of Computer Science
and Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel
michal.gordon@weizmann.ac.il

Assaf Marron
Dept. of Computer Science
and Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel
assaf.marron@weizmann.ac.il

Orni Meerbaum-Salant
Dept. of Science Teaching
Weizmann Institute of Science
Rehovot 76100, Israel
orni.meerbaum-salant@weizmann.ac.il

ABSTRACT

Scenario-based programming is an approach to software development which calls for developing independent software modules to describe different behaviors that a system should or should not follow, and then coordinating the interwoven execution of these modules at run time. We show that patterns previously shown to exist in programs written in the Scratch environment, which is not specifically scenario oriented, by children who did not have other training, and were not guided to write in a scenario-based manner, are also characteristic to scenario-based programming. These patterns include extremely fine-grain decomposition and bottom-up development. This result suggests that scenario-based programming concepts are "natural" in some ways. Thus, with an appropriate environment and a matching set of tools, scenario-based programming concepts could have an important role in early-stage computer-programming curricula.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer and Information Science Education - Computer Science Education; D.1.m [**Software**]: Programming Techniques - Miscellaneous; D.3.3 [**Software**]: Programming Languages - Language Constructs and Features

General Terms

Human Factors, Design, Languages

Keywords

Scenario-based Programming, Scratch, behavioral programming, rule based systems, aspect-oriented programming, live sequence charts, LSC, BPJ, natural programming, habits

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE'12, July 3–5, 2012, Haifa, Israel.
Copyright 2012 ACM 978-1-4503-1246-2/12/07 ...$10.00.

1. INTRODUCTION

Scratch [18] is a visual programming environment intended "...to develop an approach for programming that would appeal to people who hadn't previously imagined themselves as programmers..." (Resnick et al [22]). While drawing ideas and inspiration from a number of other programming environments designed for young people or novice programmers, Scratch is designed to be "more tinkerable, more meaningful, and more social than other programming environments" [22].

Scenario-based programming is an approach to software development which calls for developing independent software modules to describe different behaviors that the system should or should not follow, and then coordinating the interwoven execution of these modules at run time. Implementations (or manifestations) of various facets of scenario-based programming are seen, e.g., in rule-based systems such as [14], aspect-oriented programming [15], the language of live sequence charts (LSC) [5, 10], and the package BPJ for behavioral programming in Java [11].

This paper reports on qualitative research carried out by re-interpreting pre-existing data reported by Meerbaum-Salant, Armoni and Ben-Ari in [21]. We find that patterns observed in Scratch programming are significant to the study of scenario-based programming — with a focus on traits related to naturalness of language constructs.

Meerbaum-Salant et al. [21] draw attention to emerging properties of Scratch projects written by children. They studied novices who used the Scratch environment and were not guided to write in a scenario-based programming approach. Their subjects were drawn from two middle school classes (46 students, ninth grade, ages 14-15 years old, boys and girls). Each class took place in one two-hour period a week for one semester. The teachers encountered the Scratch environment for the first time; one of them was experienced in CS, the other was not. The teachers received Scratch teaching materials as well as technical and pedagogical support. The researchers did not intervene in the actual teaching of the classes. Their investigation was based primarily on four sources of qualitative data: (a) documentations of class observations (b) analysis of students' projects and exams; (c) interviews with ten students and two teachers, (d) a discussion in a focus group of two students. During the data collection and its qualitative analysis, interesting findings arose serendipitously and appeared repeatedly in different types of data. For more details see [21].

The main patterns were defined as *extremely fine-grain programming (EFGP)* and *bottom-up development.* These patterns appear to be counter to some accepted programming design practices such as top-down design, and lead to a discussion of whether training in Scratch contributes or is detrimental to the teaching of principles of programming and computer science. We examine the patterns identified in [21] and the data supporting them, and observe that the patterns are shared with applications that are designed with *scenario-based programming* in mind. The fact that these patterns were found in Scratch programs of inexperienced students suggests that scenario-based programming concepts are "natural".

When discussing *natural* behavior or patterns throughout this paper we refer to the Merriam-Webster dictionary definitions of "produced by nature: not artificial " or "not cultivated". i.e., behaviors that are either inborn, or are acquired as part of normal and general child development, as opposed to specific training in computer programming. It appears that people describe behaviors using multiple scenarios, naturally, e.g., in sharing a cooking recipe: 'I do A and then B", "It is important to do C before doing D", and "If you see E don't do F". We will show in the following sections that scenario-based programming in Scratch was natural and intuitive to the students in the sense that Scratch constructs and operators were readily understood, and seem to connect to the tendency for scenario-based descriptions. The programming patterns we describe were found repetitively and in many student works, despite not being taught or discussed explicitly by teachers, hence we consider them natural.

Meerbaum-Salant et al. [21] expressed concern about the ability of students to deal with a multitude of simultaneous scenarios, and referred by analogy to the "go-to" debate and the emergence of *spaghetti* code from inappropriate or unwise use of available commands. However, we suggest to consider the extremely fine-grained programs not as spaghetti but rather as an application of scenario-based programming. In this case we can consider explicitly teaching these concepts in early-stage computer-science curricula, thus using tools from scenario-based programming platforms as utensils for enjoying the spaghetti meal. More research is required on how to associate this with current curricula.

This paper is structured as follows. Section 2 provides background on scenario-based programming. In Section 3 we examine the programming patterns identified in [21], as well as some additional characteristics in raw data of that experiment and re-interpret their significance in the context of scenario-based programming. In Sections 4 and 5 we discuss possible implications of these findings to programming education and to software engineering in general.

2. SCENARIO-BASED PROGRAMMING

The term *scenario* is commonly used in the context of requirements specification and use cases. In this paper we focus on executable scenarios - program parts or modules which can be run as part of the application, and are aligned with the system's behavior under a certain set of conditions. There is a variety of platforms and development environments which enable the developer to use scenarios not only in early design stages, but throughout the programming process.

In *rule-based systems* one codes a set of conditions, and associates each condition with one or more required actions. The infrastructure repeatedly evaluates all conditions and for any condition that is satisfied, it executes the actions associated with it. Figure 1 depicts some of the rules in a rule-based system for controlling a boiler in a steam generator [14].

#	IF			THEN		THEN	
	dt_Level	Status	WaterLevel	DoS	DrainValve	DoS	FeedValve
1		TankFill	AboveNormal	1.00	Closed	1.00	Closed
2		TankFill	BelowFull	1.00	Closed	1.00	OpMax
3		TankFill	StartUp	1.00	Closed	1.00	MinFlow
4		TankFill	Full	1.00	Closed	1.00	Closed
5		TankFill	CritFull	1.00	Open	1.00	Closed
6		StartPump	Full	1.00	Closed	1.00	Closed
7		StartPump	CritFull	0.70	Open	1.00	Closed
8		BoilerFill	CritEmpty	1.00	Closed	1.00	MaxFlow
9	StrongDecline	BoilerFill	Low	1.00	Closed	1.00	MinFlow
10	Decline	BoilerFill	Low	1.00	Medium	1.00	MinFlow
11	Steady	BoilerFill	Low	1.00	Open	1.00	MinFlow

Figure 1: Scenario-based programming in a rule-engine: rules in a steam-generation control system. The conditions involve a status variable, the water level and its rate of change, and the actions involve opening and closing feed and drain valves. E.g., rule number 5 requires that when filling up the tank, if the level becomes critically full, the drain valve must be opened.

In *aspect-oriented programming* (AOP) [15], the programmer uses the programming language to dictate when certain modules called *aspects* should be activated, by associating them with *join points* in the base code. Aspects and base code are interwoven such that at run time when a defined join point is reached, the relevant aspect code can be executed before, after, or instead of the original code. AOP was designed to enable implementation of cross-cutting concerns in a single module, rather than the previously standard practice of inserting appropriate method calls in every affected point in the base code. AOP is available as extensions of many programming languages, most notably ASPECTJ for Java.

Behavior-based architectures for constructing systems from desired behaviors were proposed in the context of robotics and hybrid-control. These include Brooks's subsumption architecture [4], Branicky's behavioral programming [3], and leJOS [16] (see the review in [1]).

The language of *Live Sequence Charts* (LSC) [5, 10] is a visual programming language in which behavior is described in multiple separate scenarios. Each scenario (see example in Figure 2) is a sequence chart enhanced with modalities that specify which events may, must or must not happen. The chart modalities play a role that corresponds to that of the modal verbs in natural language such as *shall, can, may or cannot.* Each chart describes a piece of behavior, or "story" that may involve multiple objects. A scenario is composed of a prechart and a main chart, where if the events in the prechart occur in the specified order, then the sequence of events specified in the main chart is executed by the system. During execution, all the charts are synchronized and coordinated: e.g., whenever conditions of precharts are satisfied, main charts are activated, and events that must be triggered are indeed triggered in the specified order if they are not forbidden by other charts. The LSC language enables interwoven execution (*play-out*) of scenarios each of which may involve sequences of conditions and events in multiple ob-

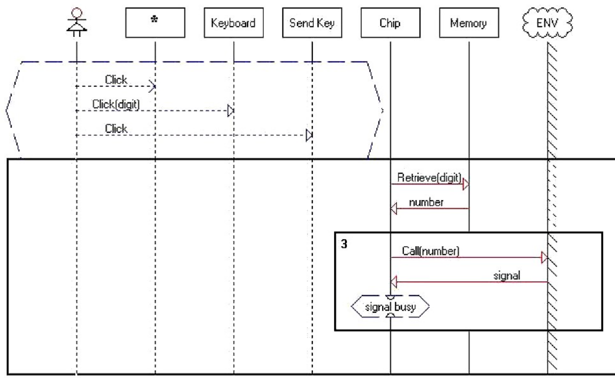


Figure 2: **Scenario-based programming example in LSC: Whenever a telephone user presses the sequence of a star, a digit and send (see hexagonal prechart), the chip must retrieve the corresponding number from memory and call it by sending a message to the environment. If a busy signal is returned, the call must be tried up to three times.**

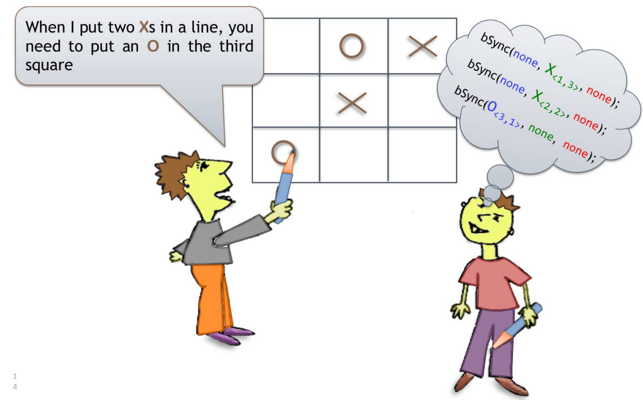


Figure 3: **Scenario-based programming in Java with BPJ: Each behavior is coded as a separate Java thread. In the child's (and the application's) mind this part of strategy needs to wait for the events X(1,3) and X(2,2) (in green), and then attempt to trigger the event O(3,1) (in blue).**

jects. Inter-scenario dependencies are solved automatically, without requiring each scenario to be aware of the others. A feature called *play-in* enables the programmer to specify the required events by acting them out on a mock-up GUI of the application (e.g. *pressing a send button* on a simulated phone display).

Behavioral programming in Java (BPJ) [11] implements the principles of LSC in the realm of standard, procedural languages, and enables their integration with standard object-oriented programming. Each behavior is coded as a Java thread, independent of the others, and all threads are synchronized and coordinated at run time to produce integrated system behavior. For example, in an application that plays the game of Tic-Tac-Toe, each of the game rules and each playing strategy is coded in its own separate behavior thread. Figure 3 illustrates how, using BPJ, individual strategies conceived in one's mind can be coded directly as independent Java modules, to be interlaced by the infrastructure during execution.

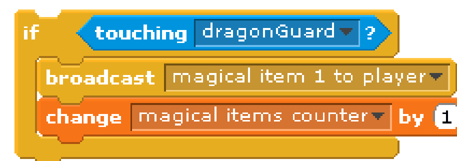
A key feature of scenario-based programming is its alignment with the requirements, or the descriptions of the system behavior as perceived by humans. This feature can be observed from different angles. Structurally, the application (i.e. its modules) can be mapped to individual requirements, as opposed to having each module responsible for parts of different requirements, as is common in standard programming. From the time dimension, looking at the development process, a scenario-based design often enables incrementality. That is, adding, refining, or removing behaviors can be done by adding new modules (rules, aspects, scenarios, behavior threads, etc.) that influence the already-developed parts of the application. See [8] for a discussion of how such features can contribute to making the development process natural, intuitive, and fun.

3. ANOTHER LOOK AT SCRATCH PROGRAMMING PATTERNS

Meerbaum-Salant et al. [21] identified two programming habits, EFGP and Bottom-Up programming, that appeared repeatedly in the students' work and in the resulting projects. These habits emerged conspicuously during the data collection phase of an investigation of learning computer science concepts in Scratch [20]. The qualitative assessment in [21] focuses on how these behaviors of the students may affect the process of learning. Given that Scratch is not explicitly or inherently a scenario-based programming environment and that this concept was not one of the basic principles that were taught in the observed classes, we propose to use the emerging patterns to gain insight about scenario-based programming and people's tendencies in this context. For simplicity and clarity of the presentation, we use a few succinct code examples which represent what was observed in different forms in many students' projects.

3.1 Extremely Fine-Grained Programming

The pattern called EFGP in [21] revolves first around the fact that the programming modules are small, and second, around the fact that composite behaviors were decomposed into separate units, instead of being handled in a single module. For example, in a game application the player has to fight dragon guards. Every time the player touches a guard with his or her sword, the player obtains a magical item that was watched over by the guard. When the player collects six items, he or she can move to the next level. In standard programming, a script to handle the event of the player overcoming a guard would often be composed of the following steps: (a) move the magical item to the player's bag (e.g., by sending an appropriate message to the item); (b) update a counter of items in the player's bag; (c) if the counter is equal to six, move to the next level. The manifestation of EFGP shows that students broke the module into at least two parts:



in script 1, and

in script 2.

In the following subsections we discuss our observations relative to this decomposition

3.1.1 Naturalness

The extremely fine-grained structure reflects the ability of Scratch developers to align code modules with the requirements or with other natural human description of the system. As the students in the referenced experiment [21] were not required to document their projects, we rely on the common observation that game descriptions provide each of the basic rules of the game separately and independently, e.g., valid moves, how one can attack, defend and earn points, what are the winning conditions, etc. Sometimes, almost as an after-thought, these descriptions include exceptions, such as, what happens when the player gets a certain number of points (e.g., go up a level), or when a certain condition arises (e.g., a draw). The possibility in Scratch to create multiple independent modules was an opportunity for the students to separate the descriptions into different scripts. This is in contrast to standard programming practices where these behaviors are likely to be coded into a single or few modules, each with many conditions and branches. In some cases the program may even be structured very differently, e.g. replacing the conditions with a tree structure for usage by a minimax algorithm. The small scripts may have a confusing visual effect when floating in the scripts area in the Scratch development environment, however, the students seemed comfortable using these building blocks. We therefore observe in Scratch the ability to align the application structure with the requirements, which was discussed as key trait of scenario- based programming in Section 2.

3.1.2 Basic use of IF statements

A facet (or an enabler) of EFGP observed in [21] is the decomposition of *then* and *else* parts of an IF statement into separate scripts - one with if <condition1> then <action1 > and the other with if <not-condition1> then <action2>. This property is very much in line with the association of scenarios with "stories". In rule-based systems and in AOP, this style is the default. When coding in LSC one can write both standard conditions (with or without else blocks), and assertions - conditions with no else structure. In the latter - when the condition is not satisfied the scenario either terminates (i.e., becomes irrelevant), or is suspended until the condition is satisfied. Additionally, precharts and monitored events in LSC imply conditions that drive further actions, with no reference to else whatsoever. When the events listened for in the scenarios do not occur, the scenario simply (and naturally) remains inactive or suspended, allowing other scenarios to handle the situation. In natural language, it is common to find descriptions such as *"If a character is hit by dragon-fire it loses a life, but if it avoids the fire it earns bonus points"*. This style leads to separate scripts for each branch of the condition, with equivalent overall semantic to that of using *else*. In [21], this habit was described by a student during an interview:

Q: Can you define what is a 'condition"?

A: when I write: "If this is such and such then say move here"; "if this is such and such then go here". So if I lay down this kind of conditions, the moment I enter one situation, he goes here, and if the situation is different, it does something else.

In this context we note that when a particular action is desired following any of several conditions, many students did not program a disjunction, most likely because the topic was not covered in these classes. It is well known that disjunctions and conjunctions are indeed more complex constructs than simple conditions, and are often introduced at later stages, as reflected, e.g. in the textbook [2]. Instead, the students coded each term of the disjunction in a separate scenario: i.e., if <condition1> then <action1> and if <condition2> then <action1> — an approach which is readily accommodated in scenario-based programming and contributes to the EFGP phenomenon.

3.1.3 Preconditions and FOREVER loops

As already seen in the previous examples, many students' scripts began with conditional operations, implemented either by constant checking of variable values or by waiting for a message that is broadcast by other scripts when certain conditions are met. In contrast with more standard procedural programming where methods are called directly by other methods, and where conditions are tested mainly as part of a longer process, in scenario-based programming modules specify the conditions under which they are activated. In rule based systems the activation triggers are a complete set of conditions over the state of the system; in aspect-oriented programming they are the specifications of the join-points; and, in LSC they are the relevant scenarios' precharts - the sequences of events that trigger the main chart components of the scenarios; in BPJ this behavior is manifested by the fact that many b-threads begin by waiting for certain sequences of events.

In [21], students often performed the constant testing of conditions using the special construct of forever_if, which continually checks whether a condition is true and whenever it is, runs the blocks inside. Similar constructs are indeed an important part of the infrastructure in scenario-based programming platforms.

3.2 Bottom-up programming

Bottom-up programming is defined in [21] as starting from components which are later linked to form a larger subsystem. This pattern is observed based on examples of students dragging separate blocks and then combining them into a script. This is in line with the intention of Scratch designers as stated by Resnick et al [22] "to make bottom-up tinkerers as comfortable as planners". While these examples demonstrate bottom-up approach, in that the young programmers did not think about the structure of the entire system before beginning to program, we propose that scenarios coded early in the process can reflect also high level behaviors, or abstractions that are not necessarily aligned with classical bottom-up structures.

For example, in [21] there is a discussion of the decomposition of the repeat until loop

into the following three scripts:

In scenario-based programming this decomposition would not be viewed as three tasks that were wrongly "separated at birth" and are now "desperately" trying to collaborate to accomplish a joint goal which would normally be better accomplished by a single module. Instead, these can be viewed as three behaviors that may be intrinsic to the sprite in their own right. One behavior is the constant moving, the other is the constant sensing whether the sprite touches other objects, and the third, reacting to such touching. Further, in a design review of such an application, one may even expect a discussion of whether the broadcasting of `stop` in reaction to the touch, is the right one, and perhaps instead, the broadcast should have been of the message `touched`, and then other behaviors could translate this to a request to stop, while others could use it for other purposes, such as touching again, or verifying the identity of the touched object, or escaping.

In other words, while the process of building the system may appear as bottom-up, the program structure suggests a decomposition which is orthogonal to the bottom-up dimension. The sprite has capabilities and/or routine behaviors, and what it does with them is programmed in the body of these behaviors. In general, these independent behavioral capabilities can be at abstraction levels that are high or low, and may not be restricted to a particular level. The programmer is not necessarily programming bottom-up, but rather, one behavior after another, as he or she thinks about them. Thus, not only can there be a mapping between the modules of the application to the sections and paragraphs of the requirements document - the process of developing some application components parts may indeed follow their sequential order in the requirements document (or in a sequential mental equivalent thereof). One may conceive the developer following a requirement document that is being read out aloud (with appropriate pauses), with little or no ability to go back in the text, beyond what is in his or her memory and with little or no modification to already completed scripts. Adjacent paragraphs may describe related or unrelated behaviors at different abstraction levels.

4. POSSIBLE IMPLICATIONS TO EDUCATION

In Scratch programming, a behavior may or may not be considered "scenario-based" based on its structure and the commands it uses. Perhaps with the right set of tools (as are available in many scenario-based platforms), scenario-based programming may be a central vehicle for teaching computer programming to young students in a way that will be natural, intuitive and fun. The required tools should cover, among others, organization of code, visualization and comprehension of code, visualization of interwoven execution and verification (see, e.g., [12, 6, 19, 9]). Such tools will help the students to not only get their scripts and sprites to run as desired, but also to "see the forest for the trees". They should be able to conceive of an application as a composite artifact whose design should have certain traits such as understandability or maintainability.

Additionally, when teaching top-down design and other accepted methodologies for creation of complex systems, the teacher's awareness of the natural tendencies for "bottom-up" construction and fine-grain design, may help clarify the points being taught.

The incrementality that comes with the scenario-based design, and the executability and feedback associated with it, are thus key ingredients in enabling tinkering, which is one of the goals of Scratch [22], and in line with the constructionism approach of Papert [13].

It should be noted that Scratch is not a full scenario-based platform, and that established scenario-based techniques and approaches have additional features such as automatic synchronization, and compact idioms for one behavior to suspend or otherwise influence the execution of all other behaviors without explicit interaction between modules. In the right setting, such concepts may prove easier to teach than the raw concepts of concurrency and interprocess communication where it is the programmer's responsibility to craft all aspects of the composite behavior.

Lastly, we conjecture that new design disciplines and practices will be developed where scenario-based programming concepts are first-class citizens among the methodologies and technologies that make up successful engineering practices. These disciplines will include, how to write independent behaviors, how to organize sets of behaviors, and what kind of inter-dependencies are acceptable. Subsequently, these practices will become a standard part of programming and computer science education at all levels.

5. DISCUSSION

While it is unlikely that humans are endowed with inherent/intrinsic skills specifically design for programming computers (as we know them today), "programming"-like skills are encountered in social human behaviors from the beginning of civilizations. From the wisdom of elders passed through generations, through social norms and laws, to everyday practices around nutrition, medicine, and art, humans were and are constantly engaged in telling others, or memorizing for themselves, how to do things. In computer science, the developers of various styles of scenario-based programming claim that the proposed techniques are "natural" in some ways - a claim that seems plausible in view of the prevailing styles of instruction manuals, cookbooks, books of law, spoken narratives with morales, etc. Qualitative studies such as [7] investigate what language features may or may not be natural or intuitive for programmers. The findings of Meerbaum-Salant et al. [21] and the findings of this paper suggest that the desire to compose full "operational descriptions" from small self-standing pieces, each of

which is introduced shortly before it is needed, is manifested not only in verbal or documented descriptions, but, given a programming language that allows it, are a preferred way for constructing all or parts of the system.

We suggest that future research could shed light on what is it in scenario-based behavior description (and subsequently, programming) that is attractive to young programmers and whether the Scratch projects uncovered a pre-existing tendency, or led the students to a totally new behavior. From our experience as programming professionals and observers of novice and experienced programmers, it appears that part of the answer lies in reducing the amount of energy or resources required at any given point in time in the development process. Perhaps fine-grained, "bottom-up" development, allows one to reduce the number of pieces that one has to think about at each stage, at the cost of having more components in the final system. What we are observing may be the stages in the process described by Lochhead [17]: "before knowledge can be organized in comprehensive global structures it first must be collected piecemeal".

In summary, the seemingly unintentional new habits which are characteristics of students' Scratch programs, during early stages of learning programming skills, appear to be well aligned with key features of scenario-based programming. To the question posed in [21], namely, that *"why the control structures were not used in the ways they were designed to be used"* this paper proposes that the answer may be that the structures which were used instead are in some way natural.

6. ACKNOWLEDGMENTS

The authors thank Michal Armoni, Moti Ben-Ari, David Harel, and Gera Weiss for their valuable comments and suggestions. The reseasrch of Michal Gordon and of Assaf Marron was supported by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013). The research of Orni Meerbaum-Salant was partially supported by the Israel Science Foundation grant 09/1277 and by a Sir Charles Clore Postdoctoral Fellowship.

7. REFERENCES

[1] R. C. Arkin. *Behavior-Based Robotics.* MIT Press, 1998.

[2] M. Armoni and M. Ben-Ari. *Computer Science Concepts in Scratch.* Weizmann Institute of Science, 2010. (In Hebrew).

[3] M. Branicky. Behavioral Programming. In *Working notes AAAI spring symp. on hybrid systems and AI*, 1999.

[4] R. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE J. of Robotics and Automation*, 2(1), 1986.

[5] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1), 2001.

[6] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On Visualization and Comprehension of Scenario-Based Programs. *Int. Conf. on Program Comprehension (ICPC)*, 2011.

[7] I. Hadar and U. Leron. How Intuitive is Object-Oriented Design? *Commun. ACM*, 51:41–46, 2008.

[8] D. Harel. Can Programming Be Liberated, Period? *IEEE Computer*, 41(1), 2008.

[9] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. In *Int. Conf. on Embedded Software (EMSOFT)*, 2011.

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

[11] D. Harel, A. Marron, and G. Weiss. Programming Coordinated Scenarios in Java. In *24th European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[12] D. Harel and I. Segall. Visualizing Inter-Dependencies Between Scenarios. In *Proc. ACM Symp. on Software Visualization (SOFTVIS)*, pages 145–153. ACM, 2008.

[13] I. Harel and S. Papert. *Constructionism.* Ablex Publishing, 1991.

[14] INFORM Gmbh. *fuzzyTECH* Software Package `www.inform-ac.com/fuzzytech.htm`.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[16] LEJOS. Java for LEGO Mindstorms. http://lejos.sourceforge.net/.

[17] J. Lochhead. Some pieces of the puzzle. *Constructivism in the computer age. Lawrence Erlbaum Associates, Hillsdale, NJ*, pages 71–82, 1988.

[18] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A Sneak Preview [Education]. In *Creating, Connecting and Collaborating through Computing, 2004. Proc. Second Int. Conf. on*, pages 104–109. IEEE, 2004.

[19] S. Maoz and D. Harel. On Tracing Reactive Systems. *Software and Systems Modeling*, pages 1–22, 2010.

[20] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Learning Computer Science Concepts with Scratch. In *Proc. of the Sixth Int. Workshop on Computing Education Research*, pages 69–76. ACM, 2010.

[21] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of Programming in Scratch. In *Proc. of the 16th Annual Joint Conf. on Innovation and Technology in Computer Science Education (ITICSE)*, pages 168–172. ACM, 2011.

[22] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Comm. of the ACM*, 52(11):60–67, 2009.