Informatics in Education, 2016, Vol. 15, No. 1, 1–23

DOI: 10.15388/infedu.2016.01

Teaching Nondeterminism Through Programming[1,2]

Giora ALEXANDRON, Michal ARMONI, Michal GORDON, David HAREL
The Weizmann Institute of Science
Rehovot, Israel
e-mail: giora@mit.edu, {michal.armoni, michal.gordon, david.harel}@weizmann.ac.il

Received: March 2015

Abstract. Nondeterminism (ND) is a fundamental concept in computer science, and comes in two main flavors. One is the kind of ND that appears in automata theory and formal languages, and is the one that students are usually introduced to. It is known to be hard to teach. We present here a study, in which we introduced students to the second kind of ND, which we term *operative*. This kind of ND is quite different from the first one. It appears in nondeterministic programming languages and in the context of concurrent and distributed programming. We study how high-school students understand operative ND after learning the nondeterministic programming language of *live sequence charts* (LSC). To assess students' learning, we used a two-dimensional taxonomy that is based upon the SOLO and the Bloom taxonomies. Our findings show that after a semestrial course on LSC, high-school students with no previous experience with ND of either type, understood operative ND on a level that allowed them to create and execute programs that included nondeterminism on various levels and in various degrees of complexity. We believe that it is important to expose students to the two types of ND, especially as ND has become a very prominent characteristic of computerized systems. Our findings suggest that students can reach a significant understanding of operative ND when the concept is introduced in the context of a programming course.

Keywords: nondeterminism, project-based learning, live sequence charts.

1. Introduction

Nondeterminism (ND) is a fundamental concept in computer science. In Schwill's work on fundamental ideas of computer science (1994), it is listed under the category of *programming concepts*. As a fundamental idea, ND appears in various domains and contexts, falling into what Schwill referred to as the *horizontal* criterion. ND also meets the

[1] A summarized version of this paper appeared in the *Proceedings of the 8th Workshop in Primary and Secondary Computing Education (WiPSCE)*, 2013.

[2] This research was partially supported by an Advanced Research Grant from the European Research Council (ERC) under the European Communitys 7th Framework Programme (FP7/2007–2013) and by the Israel Science Foundation. The work of the first author was supported by a grant from the Azrieli Foundation.

other three criteria for fundamental ideas that Schwill defined: It can be taught on various levels of complexity (the *vertical* criterion); its historical development can be traced back in the historical development of the discipline (the *time* criterion), in this case, to Rabin and Scott's work on nondeterministic finite automata (1959), for which they were given the Turing Award; and, as a fundamental characteristic of the world, it is related to everyday language and thinking (the *sense* criterion).

ND appears in CC2001 (IEEE/ACM, 2001), but is not given much weight there, and is covered mainly in the elective unit on automata theory (AL7). There can be several reasons behind the low presence of ND in that curriculum, but we believe that a major factor is that ND was far less a prominent characteristic of computerized systems at the time that curriculum was developed. However, this has changed, as reflected, for example, in a new knowledge area on parallel and distributed computing that was added in CC2013 (IEEE/ACM, 2013). The rationale that underlies this new unit is that "Given the vastly increased importance of parallel and distributed computing, it seemed crucial to identify essential concepts in this area and to promote those topics to the core" [pg. 32]. ND, especially of the kind considered in this paper, is one of these essential concepts.

In high-school, ND is probably out of the scope of most learning programs (Armoni and Gal-Ezer, 2006). (An example of a high-school program that does include ND is the Israeli high-school curriculum (Gal-Ezer *et al.*, 1995).) At the high-school level, the relative importance of the concept plays a role, but also its complexity, as high-school programs usually deal with less advanced topics. However, if we adopt the rationale of CC2013, then concepts such as ND should be introduced on the less advanced levels too. As suggested here, this can be done in the context of a programming course, an approach that has two apparent advantages. First, it exposes the students to the kind of ND that appears in parallel, asynchronous and distributed computing. Second, the context of a practical, hands-on programming course is especially appropriate for introductory level students. This concurs with the *spiral curriculum* approach of Bruner (1960), according to which, fundamental ideas should be revisited repeatedly within the curriculum and throughout the years, each time delving deeper into them.

We note that some researchers view different appearances of ND as manifestations of the same thing (this viewpoint is expressed for example in (Armoni and Ben-Ari, 2009; Dijkstra, 1976)), but they can also be seen as different types of ND. Here we distinguish between two very different types of ND. The first is the one that appears in the context of automata theory and formal languages (through nondeterministic automata). On this type of ND there is a predetermined criterion for the success of the computation, according to which it can be decided whether a specific computation is accepted or not. The second type of ND appears in the context of concurrent and asynchronous systems, and in nondeterministic programming languages. Examples of such languages include Dijkstra's *guarded commands* (1975); *logic programming* languages, such as Prolog, usually have a nondeterministic semantics (though Prolog's search rule actually makes its behavior deterministic); some modeling languages, such as Promela (which is mainly used for model checking), or *scenario-based* programming languages, such as *live sequence charts* (LSC) (Damm and Harel, 2001; Harel and Marelly, 2003a) (which is mainly a language for reactive system development), also have a nondeterministic semantics. At

the core of this type of ND lies the idea of true *don't care*, which means that there is a-priori no preference and all the possible computations are equally good. We refer to this second type as *operative* ND.

The main dichotomy between the two types of ND is as follows: the semantics of operative ND is universal by nature, in contrast to the existential semantics of Rabin and Scott. That is, in the latter case at least one of the nondeterministic possibilities is to be taken (and must lead to success for accepting machines) and in the former all are to be taken. Some difference between the two can be also captured by using the taxonomy proposed by Armoni and Ben-Ari (2009). Operative ND is meant to be run on a machine, and is consistent in terms of success/failure, but not necessarily in terms of the specific output. However, while this dichotomy tries to capture the main difference between the basic forms of these types of ND without delving into a theoretical (and controversial) discussion, it is important to mention that the issue is much more subtle. A thorough review of the different approaches to the connection between the semantics and execution models of ND programming languages can be found in (Armani and Ben-Ari, 2009). Specifically, we refer the reader to the opposing approaches of Hare and Pratt, and of Dijkstra.

In practice, it seems that students are usually introduced to ND in the context of automata theory and formal languages (Armani and Gal-Ewer, 2006), so they mostly meet ND of the first type. This type of ND is known to be difficult for teaching and learning. According to the findings of Armani and Gal-Ewer (2007), students avoid using ND in the context of computational models even when the nondeterministic automata they are asked to build are simpler than the deterministic ones, and when they do use ND, they show difficulties in creating quality solutions. Among other factors, students' difficulties are related to the high abstraction level of the concept. This concurs with other findings reported by Armani and Gal-Ewer (2004), which show that students perform better on the technical aspects than on the theoretical and more abstract aspects of the computational models course. Students' attitudes also play a role. According to (Armani and Gal-Ewer, 2007), students sometimes avoid using ND because they perceive nondeterministic solutions as illegitimate. The teaching is another factor, of course. According to (Armani *et al.*, 2008), ND is sometimes presented in automata theory course in a way that can be interpreted by the students as predictable and consistent (i.e., deterministic). Finally, Armani and Ben-Ari (2009) state that since ND is not always introduced in various contexts, its learning in the context of computational models can lead to a narrow perception of the concept, as something that is relevant only in this domain. To overcome these, the aforementioned researchers suggest to introduce ND early in the curriculum and according to the spiral principle, and to focus on the design of ND automata by the students themselves.

With respect to the learning of operative ND, Ben-David Kolkata (2004) identified the ND inherent in the execution of concurrent programs as the main source of difficulty in learning this subject.

There are studies that address both types of ND (without necessarily distinguishing between them). Giant reported on difficulties in perceiving the concept of *don't-care* (2010), as it appears in the context of solving algorithmic problems. He related the

problems found to the ability to move from a concrete to a more abstract perspective, to accept an arbitrary ordering or a random starting point, and more. Ben-Ari (2010) described an interesting approach for teaching concurrency and ND of the first type, using a self-developed tool that is based upon the SPIN model checker.

We concur with the rationale of CC2013, and believe that it is important to introduce students to both types of ND. In order to evaluate the feasibility of teaching operative ND on the high-school level, we examined the learning process of high-school students that were introduced to this concept in the context of a programming course on LSC and scenario-based programming. Since operative ND is tightly connected to concurrent and asynchronous programming, it is natural to introduce it in the context of such a course, as this kind of programming is inherent to LSC. Furthermore, we believe that such a course should emphasize learning-by-doing. Numerous sources, e.g. (Schank *et al.*, 1999), indicated that when engaged in learning-by-doing, also known as hands-on learning, students gain better and lasting retention of the learned material.

To achieve this, our course followed two pedagogic principles – the "zipper principle" (Gal-Ezer *et al.*, 1995), and project-based learning. The zipper principle means that theoretical lectures are interweaved with hands-on experience in the lab, in which the students exercise the learned concepts on a small scale and in a controlled setting. This supports gradual, bottom-up learning of the basic programming constructs and the execution model. Project-based learning (Blumenfeld *et al.*, 1991) is basically a top-down learning approach. Students start from what they want to build, and use their knowledge (and if needed, acquire additional knowledge) in order to realize it. This requires the learners to synthesize their knowledge, gives it a real-world context, and emphasizes collaborative work. Among other things, project-based learning also increases motivation and engagement. A more detailed description of the course and the pedagogic rationale that underlies it will be published separately.

Based upon the results of the research that we present below, we believe that high-school students can indeed reach a significant understanding of operative ND. One advantage of our approach is that it does not require adding an isolated topic to the curriculum. It can be implemented by extending an existing programming course with the new concept. To enable this, the main issue is to choose the appropriate language.

The rest of the paper is organized as follows. In Section 2 we briefly describe LSC. In Section 3 we present the research question, the methodology, and the findings. In Section 4 we discuss the findings, and in Section 5 we present our conclusions.

2. Live Sequence Charts

In this section we briefly describe the language of *live-sequence charts* (LSC) and its development environment, the *Play-Engine*. The language was originally introduced in (Damm and Harel, 2001) and was extended significantly in (Harel and Marelly, 2003a) and (Harel and Marelly, 2003b). LSC is a visual specification language for reactive system development. LSC and the Play-Engine are based on three main concepts, which we now briefly review.

2.1. *Scenario-Based Programming*

LSC introduces a new paradigm, termed *scenario-based programming*, implemented in a language that uses visual, diagrammatic syntax. The main decomposition tool that the language offers is the *scenario*. In the abstract sense, a scenario describes a series of actions that compose a certain functionality of the system, and may include possible, necessary or forbidden actions.

Syntactically, a scenario is implemented in a live sequence chart. An example is shown in Fig. 1[3].

A chart is composed of two parts – the *pre-chart*, and the *main-chart*. The pre-chart is the upper dashed-line hexagon, and it is the activation condition of the chart. In case that the events in the pre-chart occur, the chart is activated. Execution then enters the main chart. This is the lower rectangle, which contains the execution instructions. The vertical lines represent the objects, and the horizontal arrows represent interactions between them. The flow of time is top down. The chart in the example describes a simple scenario taken from the implementation of a cruise control. Once the user presses the brake pedal, the cruise unit releases control of the brake and the accelerator, and then turns itself off.

2.2. *The Play-In Method*

LSC is supplemented with a method for building the scenario-based specification over a real or a mock-up GUI of the system – the *play-in* method (Harel, 2000; Harel and Marelly, 2003a,b) – which is implemented in the Play-Engine. With play-in, the user specifies the scenarios in a way that is close to how real interaction with the system occurs. This is illustrated in Fig. 2[4]. The figure shows a GUI of a cellular phone, and a simple LSC diagram containing a scenario that describes what the display and the speaker

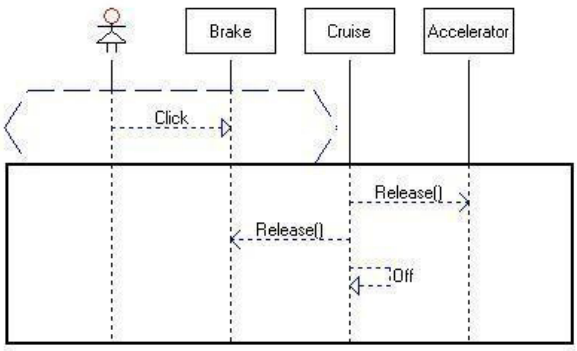

Fig. 1. LSC chart[3].

[3] Figure Fig. 1 reproduced from (Alexandron *et al.*, 2013)

[4] Figure Fig. 2 reproduced from (Alexandron *et al.*, 2014)

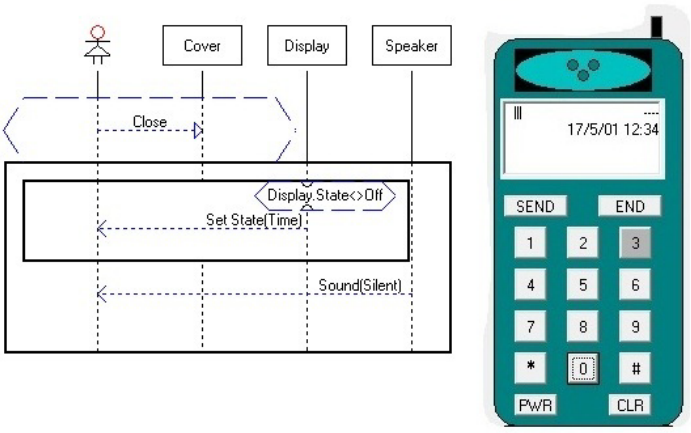

Fig. 2. The play-in method[4].

should do once the user closes the cover of the cell phone. The GUI and the scenario are taken from the examples that are supplied with Play-Engine (Harel and Marelly, 2003a). The focus here is on how this scenario was 'programmed' into the system. This was done by clicking on the components of the GUI (i.e., by *playing* with the GUI).

One of the motivations behind this 'programming through the interface' approach is to allow people who are not familiar with LSC (or even with other programming languages), to program the behavior of an artifact, or parts of it relatively easily.

2.3. *The Play-Out Method*

LSC has an operational semantics that is implemented by the play-out method (originally introduced in (Harel and Marelly, 2003b)). It too is included in the Play-Engine. Play-out makes the specification directly executable/simulatable. When simulating the behavior, the programmer is responsible for carrying out the actions of the potential end-user and of the system environment. Play-out keeps track of the user/external actions, and responds to them according to the specification. The play-out algorithm interacts with the GUI to reflect the state of the system on the y. For more details see (Harel and Marelly, 2003a).

2.4. *Nondeterminism in LSC*

LSC is a nondeterministic programming language. At its core, the kind of ND that is inherent in the language stems from the idea of *don't care*: At each point of the computation, there are some branches that can be taken, where all the branches are equally good. To achieve that, the language supplies mechanisms that allow to describe various aspects of the system behavior without being forced to introduce determinism into the implementation, when it is not derived from the requirements. Here we briefly review some of these mechanisms. For more details see (Harel and Marelly, 2003a).

One example is LSC's partial order scheme. In LSC, the default ordering (in and between charts) is nondeterministic, but in several cases (defined by the partial ordering rules) the order of execution between some of the events is mandatory. An example can be seen in Fig. 3, in which the order between the two assignments of Key.Value into to X1 and X2 is nondeterministic, but due to the *sync* event (the dashed-line hexagon), these two events must happen before the 'Show' event, i.e., in a deterministic order.

Another example is LSC's existential binding of *symbolic instances*, which allows stating that a certain activity should be done by *some* object, without having to state *which* object. An example can be seen in the left chart in Fig. 4. The 'Key::' object (the upper dashed-line rectangle) represents any of the 1–9 digit keys.

Other constructs include the *must vs. may* modality, which allows stating that a certain behavior is optional, meaning that it may or may not be executed, depending on external (thus, nondeterministic) conditions; *select*, which is a weighted *case* construct, with the weights defining the probability of choosing each of the branches; and more.

As a model of computation, LSC also includes parts that implement an asynchronous, thus nondeterministic, semantics. For example, this is the semantics of asynchronous messages.

In the course, the students were introduced on various levels to all the above, except for asynchronous messages.

3. The Study

The research question that we studied is how high-school students understand operative ND after learning the language of LSC. The motivation was to check whether students can reach a significant understanding of operative ND after learning LSC, as learning ND in its various flavors in known to be hard for students (see Section 1). As discussed below, the students had no previous background or knowledge regarding nondeterminism, and we were not interested in checking whether teaching operative ND through LCS is more effective than teaching operative ND with other possible methods. Thus, we did not use a comparative design or a pre-post design. Rather, we assessed students' knowledge during and after the teaching process. This approach is inline with other studies that focus on how certain methods can be used to achieve desired learning objectives. One example is the work of Meerbaum-Salant, Armoni and Ben-Ari (2010), who studied how the Scratch programming language can be used for teaching basic computer science concepts.

The study was carried out on the basis of a course on LSC and its underlying paradigm, scenario-based programming. The collected data included exam questions, programming projects given as a final assignment, and post-interviews held with representative students. To assess students' learning, we used a two-dimensional taxonomy and a combined quantitative and qualitative methodology.

This section is organized as follows. First we describe the research setting. Second we describe the analysis tools and process. Third we present the findings.

3.1. *The Research Setting*

The context of the study was a forty-five hours (three hours per week) semestrial course given to nineteen 12th grade (age: 17–18) high-school students majoring in computer science (gender makeup: 10 girls, 9 boys). The course was developed and executed as a pilot course aimed at teaching scenario-based programming and reactive systems development with LSC, and it was mandatory for the students of the class that was chosen for the experiment. As described in Section 1, the course structure was arranged according to two pedagogic principles – the "zipper principle" (Gal-Ezer *et al.*, 1995) (the first half of the course), and project-based learning (the second half of the course). The projects were developed in groups, under the guidance and assistance of the teacher. There were five groups, each of three or four students. The project was to implement a reactive system, and the students could either choose the system by themselves, or choose something from a list that the teacher prepared. Students' projects included implementing a memory game ("Simon"), modeling the behavior of an elevator, etc. The students were also given written exams in which they were required to comprehend and modify systems (or parts of systems) implemented in LSC.

3.1.1. *Students' Previous Experience*

Our students experience in computer science included two introductory computing courses and a course on computer organization and assembly languages, taken in grades 10 and 11. During the first semester of grade 12, the students took a course on computational models. In order to allow the instruction of the LSC course in the second semester of this grade, the computational models course, that usually spans over the entire year (two semesters), was shortened to fit into one semester and thus did not include nondeterministic models. Also, during this grade the students took a yearly course on software design. In none of these courses they were introduced to the concept of ND or concurrent programming.

Teaching ND was also not an explicit objective of the LSC course, but an issue that emerged due to the need to explain the nature of LSC and due the fact that the students were not familiar with the concept. Thus, the teacher gave the students a formal definition of the concept at the beginning of the course, and referred to this definition afterwards when discussing nondeterministic semantic issues (see section 2.4).

3.2. *Assessing Students' Understanding*

To assess students' learning, we used a two-dimensional taxonomy, which can be viewed as a table. Each cell in the taxonomy represents a certain level of learning. By analyzing the data, we estimate the extent to which each level of learning was achieved. Our taxonomy is a slight variation of the two-dimensional taxonomy of Meerbaum-Salant *et al.* (2010), which was developed to assess students' learning of computer science concepts. It is built upon two existing taxonomies. The vertical axis is based upon Bloom's taxonomy in its revised form (Anderson *et al.*, 2001), and the horizontal axis is based upon

the SOLO taxonomy (Biggs and Collis, 1982). The taxonomy is presented in Table 1[5]. The table also includes the analysis approach used for each category.

We find this combination appropriate for our purpose because the Bloom and the SOLO taxonomies refer to two dimensions that grasp essential properties of the programming activity.

Bloom's taxonomy classifies knowledge according to the ability to perform actions in increasing level of cognitive complexity. From Bloom's taxonomy we chose to focus on two categories (out of six): Creating and Applying. We find this subset appropriate for our purpose due to several reasons. First, the operational interpretation of these categories in the context of programming seems relatively natural, and is straightly associated with the activities that the course focused on – the understanding of the language semantics and the creation of programs. Understanding of the semantics was operationally defined as the ability to mentally simulate or track algorithms, and was associated with *applying*. Creation of programs was interpreted as *creating* (See more on the interpretation below). Debugging, which is more naturally associated with *analysing* and *evaluating* in the revised Bloom's taxonomy, was less emphasized in the course. Second, as several authors have mentioned, Bloom's categories can overlap, and the classification of operations into categories can be ambiguous (Fuller *et al.*, 2007). Thus, we concentrate on one intermediate category (applying) and one higher category (creating). This allows us to consider the categories in a somewhat broader form, which makes the classification less ambiguous, and produces a significant amount of data in each category.

Concentrating on a subset of the categories, or using fewer meta-categories, is an accepted approach. See for example in (Lister and Leaney, 2003; Meerbaum-Salant *et al*., 2010).

The SOLO taxonomy puts the focus on the scope of the learning activity, from a local to a global and holistic perspective. We find this very natural for describing programming activities, since it is inline with the structure of computer programs which are built by composing smaller functional pieces together to obtain higher level functions. From the SOLO taxonomy we chose to focus on the three intermediate categories (out of five): Unistructural, Multistructural, and Relational. These categories have a relatively natural operationalization in the context of ND in LSC (see below). The lowest category of the original five (Prestructural) is of less interest for us, since it gives very little information about what was learned. The highest category of the original five (Extended Abstract), which is mainly about transfer, was not relevant in the context of our course and study.

Table 1

The taxonomy and the analysis conducted on each category[5]

	Unistructural	Multistrcutural	Relational
Applying	Quantitative + Qualitative	Quantitative + Qualitative	Qualitative
Creating	Quantitative + Qualitative	Qualitative	Qualitative

[5] Table reproduced from (Alexandron *et al.*, 2013)

The way we use the SOLO taxonomy is somewhat different from the common use of this taxonomy in computer science educationComputer Science Education research papers. The SOLO taxonomy is usually used for assessing students' learning, and performed by comparing students' answers to a given question versus an optimal answer to this question. The assumption is that the question can be answered in a way that reveals understanding till the highest SOLO level, so an answer that reveals understanding till a lower level indicates a problem or an insufficient learning. See for example in (Lister *et al.*, 2006).

For us, this methodology is less appropriate since it focuses on the highest level achieved, while we are also interested in the milestones and the knowledge obtained on lower levels. Thus, we looked for evidence of significant learning that occurred on several levels. This rationale was applied to the quantitative and qualitative analysis that we conducted. In the quantitative analysis, we therefore used exam questions of varying levels of complexity, and each question was classified into the highest SOLO level that this question measures. In the qualitative analysis, which was based on the projects submitted by the students and on the interview held with them, we looked for evidence for learning that occurred on each category of the taxonomy. Though this is a less common use of SOLO, it actually concurs with Biggs' ideas, for example of using the SOLO taxonomy for devising a test containing ordered-outcome items (Biggs, 1999).

The two taxonomies are hierarchical, and the categories are referred to as inclusive. However, in the combined taxonomy we do not give superiority to either of the taxonomies, so we do not assume any hierarchy between categories in addition to the hierarchy that can be derived from the individual taxonomies. This is a weaker relation than the one suggested in (Meerbaum-Salant *et al.*, 2010), which gave superiority to the SOLO taxonomy in order to have a fully ordered taxonomy. For our purpose a full order is not required.

3.2.1. *Operationalization*

As discussed in (Meerbaum-Salant *et al.*, 2010), experts find it hard to agree on the interpretation of the taxonomies for CS tasks. We follow the interpretation suggested in that study, and adapt it to LSC. This yields the following operative definitions for the atomic components of the two-dimensional taxonomy.

From Bloom's taxonomy:

- Applying: the ability to execute algorithms or code. In the context of LSC, this is the ability to track and simulate pieces of code that contain a nondeterministic element.
- Creating: the ability to plan and produce programs or algorithms. In the context of LSC, this means to implement an LSC program (or pieces of it) that contains a nondeterministic element/s.

From the SOLO taxonomy:

- Unistructural: local perspective. The interpretation to LSC means acting in the scope of a single chart. In our case, this syntactic-based criterion was enough,

since the charts were simple and short (it is possible that a long, complicated chart could not be considered as Unistructural).
- Multistructural: a perspective that incorporate multiple LSC charts.
- Relational: holistic perspective, referring the whole program or a property of it.

For example, the combined category of Creating/Relational, in the context of ND, describes the ability to create an LSC program in which ND is an essential characteristic that affects top-down design decisions. The interpretation of Creating/Multistructural is creating a program that contains ND which involves multiple charts. On this level, it is enough that ND is the result of a local programming decision or an emerging property (as long as the programmer is aware of the nondeterministic semantics of the code).

3.2.2. *Mapping the Data into the Taxonomy*

As said, our objective was to assess the level of knowledge obtained on each category of the combined taxonomy. On the more simple categories (up and left in Table 1), we used a quantitative analysis to calculate a score that represents the level of knowledge obtained. This score was based on the results of questions given in the course exams. The qualitative analysis was used to get deeper understanding of the learning outcomes, by finding evidence for learning that occurred on each of the categories. To complete the picture, we also describe problems encountered. The qualitative analysis was based on the projects and the post-interviews. The type of analysis conducted on each category is presented on Table 1.

3.3. *Findings*

We start with presenting the quantitative findings, and then continue to describe the qualitative findings. A summary of the findings is given in section 3.3.3.

3.3.1. *Quantitative Findings*

The quantitative analysis was based on the exam questions. Overall, we used four questions, one per category, except for the category of Applying/Unistructural, for which we used two questions (since this is the lowest category, more questions fitted it; the questions are described below, within the subsections describing the findings in the specific categories). The analysis was conducted as follows. First, for each category that was quantitatively assessed, we graded the question/s belonging to this category (this grading referred only to the nondeterministic elements of the question, so it was different from the exam grading). A full and accurate answer was given 100%, a partial answer was given 50%, and a wrong answer was given 0%. Finally, each category was given the average score of all answers belonging to it. Because the exam questions were elective, we could not guarantee that all the questions will be answered by the same group of students. However, since we refer to the quantitative results only as a benchmark, we preferred to include all the answers, rather than focusing only on the subset of the students that answered exactly the same questions.

To process was validated as follows. The classification of questions into categories was validated by an expert who was not part of the research team, and who classified a sample of 25% of the questions that were used. This classification was similar to ours. To verify the grading, a sample of 33% of students' answers was graded independently by two of the authors, and the grades were compared. The level of agreement was high (above 90%).

The questions that were used for each category are shown below. The results are summarized in Table 2.

Applying/Unistructural. The question contained a chart describing some scenario (taken from the specification of a calculator). The chart is shown in Fig. 3[6]. Due to the semantics of LSC, there is no mandatory order between the two actions X1:=Key.Value and X2:=Key.Value located in the lower rectangle. The students were requested to i) identify whether the code can be executed in several orders, and ii) If so, to supply two possible orders. The two sub-questions got the same weight. This question is classified as *applying* because it requires the students to execute a given algorithm, and as *unistructural* because this algorithm resides in the scope of a single chart. This question appeared as an item in a question that was mandatory, so all the students that took the test (18 out of 19) answered it. We also used in this category a question from another exam, in which the students were also required to identify possible orders within a single chart (taken from another system). This question was elective, and nine of the students answered it.

Applying/Multistructural. The question contained the LSC charts shown in Fig. 4 (the scenarios were taken from the specification of a cell-phone). The semantics is that the left-hand chart launches the event Send(Memory.Number) that activates the right-hand chart. There are several possible legal ways to carry out the combined execution, since there is no mandatory order between the actions in the right-hand chart. The students were requested to i) write the events that will be executed during the run

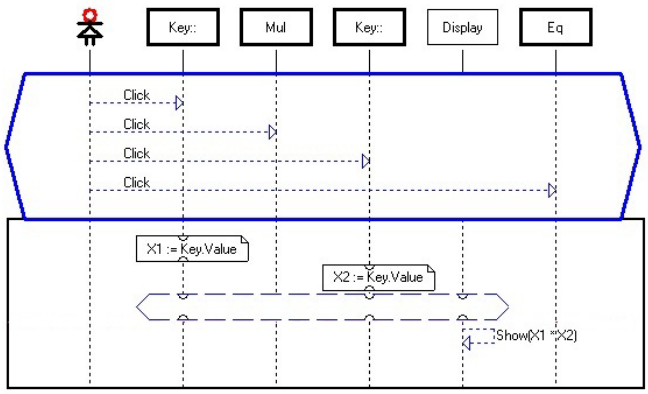

Fig. 3. Applying/Unistructural[6].

[6] Figure reproduced from (Alexandron *et al.*, 2013)

of these two diagrams; ii) identify whether the code can be executed in several orders, and if so, iii) to supply two possible orders. All the sub-questions got the same weight. This question is classified as Apply because it requires the students to execute a given algorithm, and as Multistructural because this algorithm is captured by the combination of two charts.

Creating/Unistructural The given question was as follows: "Add a scenario implementing the following requirement: When the user turns the main switch of one of the ovens off, the light and the heating element of this oven are shut down." This question includes a nondeterministic element because there is no mandatory order between the action of turning off the oven's heating element, and that of turning off its light. Thus, a proper answer to this question should include a chart that does not restrict the order between these actions and allows all legal execution paths. An example of a correct answer is shown in Fig. 5. Due to the semantics of LSC, there is no order between the two 'state(Off)' actions. We consider this question as Create because it requires the student to create a new piece of code, and as Unistructural since this is done in the local scope of a single chart. The question was elective, and was chosen by ten of the nineteen students.

However, using this question as an operational measure for the creation of nondeterministic code includes a delicate issue. Assume that a student gave a valid answer that did not restrict the order between the two actions. Does this mean that the student actually knew that the code he/she was writing contains this exibility; i.e., that its execution is nondeterministic? To make sure that we actually consider only such students, and not

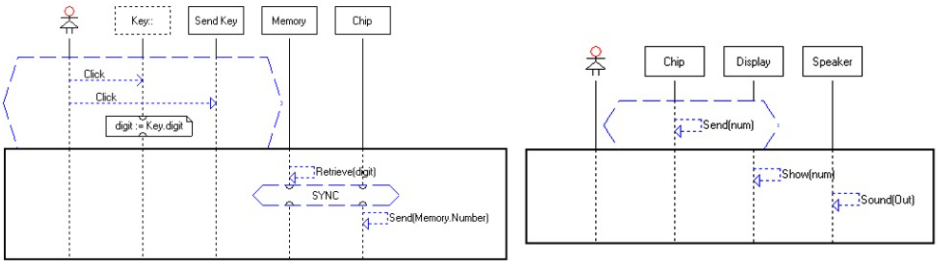

Fig. 4. Applying/Multistructural.

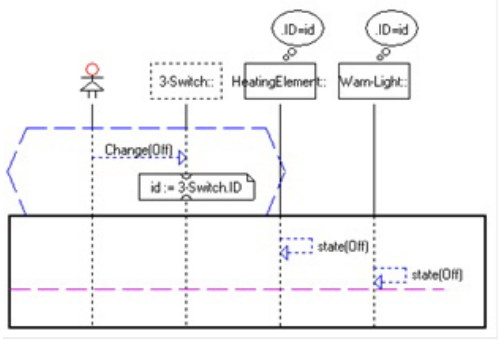

Fig. 5. Creating/Unistructural.

students that might have written this code without being aware of its nondeterministic semantics, we took only students who showed (on another task) that they can identify ND in the context of a single chart. This means students who answered the questions classified as Applying/Unistructural (see above) correctly. This process filtered out one student who answered this question correctly, but who did not correctly answer the Applying/Unistructural question. So we remained with nine students, all of whom succeeded in this question.

This issue is related to a more general dilemma, which, in turn, is related to what can be considered the 'creation of ND' in LSC. In LSC, creating nondeterministic code is sometimes easier than creating deterministic code, and creating a nondeterministic program can be sometimes done without understanding its underlying nondeterministic semantics. Thus, we had to decide whether we consider as *creating* all the cases where ND is created, or only the cases were we can confidently say that ND was consciously created and understood. We chose the more restrictive option. This interpretation forced us to exclude some of the findings, but it is more consistent with the taxonomy (a point also referred to in the discussion).

Summary of the quantitative results. Table 2[7] summarizes the score calculated for each category. N stands for the number of answers considered. As can be seen, students' achievement on these categories are satisfactory.

3.3.2. *Qualitative Findings*

The qualitative analysis was based on students' final projects in LSC, and on the post-interviews held with four representative students (one student per group, except for one group of three students that did not submit its project). With the exception of student #4, who was chosen mainly because of her enthusiasm and dominance in the team, the other students were chosen more or less at random, based on their availability.

The qualitative analysis was validated with an expert who is a CS education researcher and was not part of the research team. The expert was given one of the four projects (the Simon project), and the interview that was held with the representative of the group that created this project (student #4). The expert was then requested to analyze in depth the data, map it to the appropriate category/ies of the taxonomy, and explain the mapping. The expert's mapping was similar to ours, but she backed it up with a different argument. Overall, the process reinforced our confidence in the findings, and enriched them with another perspective.

Table 2
The quantitative results[7]

	Unistructural	Multistructural	Relational
Applying	83%, N = 26	76%, N = 18	
Creating	100%, N = 9		

[7] Table reproduced from (Alexandron *et al.*, 2013)

In general, the lower level categories (such as *Applying/Unistructural*) were relevant to all the projects. The higher level categories were relevant to some of the projects, since the nature of the projects put an upper bound on the level of dealing with ND in the project. Our findings indicated that all the students reached a sufficient level of understanding in the lower- to intermediate-level categories, and that the students who created projects that included ND on a level that fell into the higher categories, also reached a sufficient level of understanding on that levels. This is demonstrated below by examples illustrating the knowledge that was built in all the categories, together with examples illustrating some difficulties that were observed. The analysis is presented top-down, from the higher level category to the lower level one. This ordering was chosen in the purpose of facilitating the presentation, since the description of the high-level categories includes a description of the projects, which helps to understand the context in which the low-level categories reside.

Creating/Relational: This is the highest category of the taxonomy. Below we first show how learning that belongs to this level was manifested in one project, and then how the design decisions taken by the students in another project reveal some difficulties involved in understanding ND on this level.

Student #4 and her teammates developed a variation of the 'Simon' memory game. The game works as follows: The user is given a random sequence of four colors, and is then required to repeat this sequence by heart. In case of success, a new random sequence is given. The crux of the game is the nondeterministic output of the program, so creating a program that is arranged around this main theme required an abstract view and a global understanding of the role of ND in the program. The fact that the students perceived this element as the essence of their project was manifested in their design decisions. Except for its nondeterministic behavior, they did not insist on any of the other apparent characteristics of the original game (color, sound, form, increasing length of the sequence). Thus, we consider this kind of creation to be *relational*.

Obviously, this game can be implemented in various programming languages. However, it seemed that with LSC creating a program that uses ND as a programming means was straightforward for them. This is exemplified by a contradicting example. The student mentioned that she built Minesweeper game in the Assembly language course. This game also has a nondeterministic property (the location of the bombs on the board). When asked what the difference between the projects was, the student said:

> *S: I didn't make it random [the Minesweeper in Assembly], there was only one board, it was always the same game, so if you remembered the location of the bombs you could always win...*

With LSC the student did implement this basic property of the game. In fact, the student (and her teammates) implemented this property twice, using two different nondeterministic features of the language (this was due to a bug in the development environment, which the students found when debugging their system. The debugging process is described and analyzed in the next category). This process, in which the same abstract property was implemented twice using different mechanisms, reinforced our belief that

the use of ND in this project revealed a high level learning, and that this learning was related to the nondeterministic nature of LSC.

A difficulty related to learning ND on this level was revealed in the project of one of the groups. This group modeled the behavior of an elevator (we later refer to it as the 'elevator group'). However, they didn't know how to address the requirement that users on different floors can call the elevator simultaneously. To overcome this, they designed their system under the assumption that the system can receive a new request only after the previous one was handled. Though concurrency is the main issue behind this relaxation of the requirements, ND and concurrency are closely related, as we noted before. The fact that the students preferred not to deal with this requirement indicates that they had a difficulty (at least according to their subjective feeling) in creating a system that addresses it. On the other hand, it also indicates that they were able of recognizing the implications of this requirement on the complexity of the system and the scenarios it needs to handle.

The findings suggest that when introduced through a nondeterministic programming language, it might be easier for students to adopt ND as a tool in their programming 'tool box'. However, we can still expect to see difficulties related to ND in this level in the context of concurrency.

Applying/Relational: During the development process, the semantics of the language are used to develop a program that achieves the desired goals. The implementation process can be described as 'cyclic refinement through simulation' (Alexandron *et al.*, 2011): In each implementation step, the program is executed and the gap between its real behavior and its desired behavior (as expected by a mental simulation) is assessed. Then, another implementation cycle is carried out, to add functionality, fix bugs, etc. Thus, building a program involves a mental simulation of the developed artifact, and this mental activity falls into what we defined as *applying*.

This was manifested, for example, in the behavior of student #4 and her teammates. As described above, ND played a central role in the project of these students. They first tried to achieve the desired randomized behavior using one nondeterministic feature of the language – *existential symbolic instances*. However, there was a bug in the development environment that caused this feature to actually behave in a deterministic manner, so it gave the same output each time. The students were very surprised because they expected to see nondeterministic behavior, and the deterministic behavior surprised them:

> *I: In the first [attempt] you had a different solution, which uses existential symbolic instances, right?*
>
> *S: Yes, but we saw that it always randomizes the same thing, and we wanted it to be a real randomization.*

So, based on the cognitive model described above, the fact that the student had a (valid) expectation of the program's behavior is an indication of a mental simulation that in this case involves a nondeterministic property. Thus, we interpret it as *applying*. Also, the assessment process required that the students: i) Have the notion of a single execution of the program, which is composed of a sequence of events, as an instance (i.e., an object of thought; capturing an algorithm as an 'object' is considered a high level

of abstract thinking (Perrenet *et al.*, 2005)), and ii) Compare between such instances of several runs. This is a 'meta' perspective. Thus, we consider this behavior as *relational*.

A difficulty related to learning nondeterminism on this level, in the context of concurrency, was revealed in the interview with student #2, who was part of the 'elevator group'. As described in the previous section, this group built their system under the assumption that it does not need to handle concurrent calls from the user. In the interview, the student was asked to describe the system behavior in case the system indeed receives concurrent calls, and was not able to supply a sufficiently detailed answer. We interpret this task as *applying*, since the question was to mentally simulate the system behavior, and as *relational* because it requires a meta-perspective of the system (the student does not know which modules can be affected, so he needs to consider the entire system).

Creating/Multistructural: Into this category we ascribe design and/or coding of program modules that contain multiple LSC diagrams, and in which ND plays an important role. An example is given in the project of student #1 and her teammates. The student modeled a coffee machine. In this project ND served as an abstraction means, in the sense that it allowed the students to define simultaneous scenarios without committing to specific execution order between the scenarios in places where no specific order was required. The project included two diagrams that are activated simultaneously as response to a certain system event. Once the activation event is launched, either of the charts can progress, which means that the interleaving of the executed events is nondeterministic. The order of execution can very between runs, and the program is correct under all the possible orders. The students relied on the fact that the scenarios can progress simultaneously, that there is no mandatory order between them, and that this does not affect the correctness of the program:

> *I: So both scenarios can progress simultaneously?*
>
> *S: Yes.*
>
> *I: And does it matter?*
>
> *S: I don't think so.*
>
> *I: but is it something that you considered? Did you think whether the charts will be activated together or not?*
>
> *S: Yes, but they can be activated together without interfering each other [...]*

So, we see that when programming with LSC programmers learn to use ND as an abstraction means, and that the ND built into LSC helps them to reduce the cognitive load involved in programming by allowing them to ignore unnecessary implementation details such as the order of execution. However, we also saw evidence of difficulties in creating knowledge on this level. For example, student #2 showed that he is able of mentally simulating ND in the context of several charts (this example is also discussed in the next category, Applying/Multistructural), but he then mentioned that actually only one of these nondeterministic behaviors was desired, but that he did not know how to solve it. This means that the student had a problem with *creating*. However, to be more specific, the problem was with *removing* ND, not with *creating* ND.

Applying/Multistructural: We interpret a mental simulation of the program as 'Applying'. When such a mental simulation involves multiple charts, and includes a nondeterministic element, we ascribe it to the category of Applying/Multistructural. An example was given in the behavior of student #2. As described on the previous category above, the student's project included unwanted nondeterminism between two charts. In the following excerpt, the student demonstrates that he is capable of *simulating* the nondeterministic execution.

> *I: So I understand that you wanted the elevator to be shut down only after all of this [the scenario in chart E] happens, right?*
>
> *S: Yes, but I understand that there is nondeterminism here, if it opened the elevator, so this might happen before that [i.e., before the chart is over].*

So, the student understood that an execution that involves two charts can take various routes, and he was mentally able to simulate these routes. In this case, the student also specifically used the term 'nondeterminism'.

Creating/Unistructural: This level refers mainly to a programming activity that takes place in the scope of a single chart. For example, student #4 used the nondeterministic features of LSC to achieve randomization in the Simon project. As described above, the first attempt did not succeed, due to a bug in LSC's development environment. To bypass this and achieve randomization, the student used *select*, which is a different feature of LSC, which has nondeterministic semantics. Select allows the developer to supply a list of weighted choices, with the execution engine randomly picking one of the choices according to the weights. The student's description indicates that she understood this semantics and used it as the randomization core of the program:

> *I: Let's look at the diagram that does the random choice of the squares. This diagram is called 'Random'. Can you explain to me what it does?*
>
> *S: I noted that square ID's were 1 to 4, so I made it choose with a probability of 25% a number between 1 and 4, and to do it four times. Every time it chose a place [i.e., an ID], it opened another diagram that highlighted this place [i.e., the square with this ID]...*

Since this semantic expression operates in the scope of a single chart, we consider this as Create/Unistructural. Again, this is an explicit use of nondeterminism to achieve randomization. The use of nondeterminism as an abstraction means, which allows writing a scenario without committing to a specific order between (some of) the events in the scenario, was seen in all the projects.

Applying/Unistructural: This is the lowest level of knowledge that we measured. It refers to the ability to mentally simulate a piece of code that includes a nondeterministic element. As this is a relatively local and low-level operation, it was assessed mainly using the exam questions (see the quantitative analysis). However, we would like to show one example that illustrates the hierarchy of the taxonomy, and how this hierarchy cor-

responds to the operational definitions that we used. Consider the behavior of student #2 that was described in the Apply/Multistructural category: The student mentally simulated the nondeterministic execution of two charts (i.e., a Multistructural level activity) by composing the nondeterministic execution of the two single charts (i.e., an Unistructural level activity). So, the more global perspective is composed of more local perspectives, and thus contains them.

3.3.3. *Summary of Findings*

To summarize the findings, we saw that:

i). Students were able of understanding nondeterministic systems on a level that allowed them to mentally simulate parts of the systems or the systems as a whole, in a way that considered the nondeterministic element of the system. On this level we found some evidence of problems when ND was associated with a high-level of concurrency in the level of the entire system.

ii). Regarding the ability to create nondeterministic systems, we found that:

a. Almost all the students were able to create ND in the local scope of a specific module, or in the wider scope of several modules. In this context ND was usually used as an abstraction mechanism, which enabled hiding unnecessary implementation details as the order of execution.

b. Some of the students demonstrated the ability to create ND in the scope of a whole system. However, we saw some evidence of students trying to avoid dealing with ND on the level of the entire system when it was associated with a high-level of concurrency.

4. Discussion

We believe that teaching operative ND in the context of a programming course that includes hands-on experience of building and executing systems promotes meaningful learning of the concept. Our course followed two pedagogic principles, the zipper approach, and project-based learning (see Section 1).

The zipper approach (Gal-Ezer *et al.*, 1995), which emphasizes small intervals in which basic programming constructs are learned and exercised simultaneously in the lab, allows the students to build a functioning mental model of the basic components of the language and its execution model. Since even some of the most basic components of LSC have nondeterministic semantics, ND is there from the start. This creates a learning environment in which ND is the "normal situation", as suggested by Dijkstra ((Dijkstra, 1976), p. xv). When this is the case, the students can gradually develop their understanding of the concept, and along the way, get used to the idea of using it. This is opposed to the way ND is introduced in courses on automata theory, as an extension to the basic model, which requires the students to substantially extend their mental model in one leap, yielding a cognitive difficulty. It also gives the students a feeling that ND is not the normal situation, potentially causing him/her to resist accepting ND as a legitimate solution (such an attitude was reported in (Armoni and Gal-Ezer, 2007)).

Another characteristic of LSC that promotes the meaningful learning of operative ND are the various appearances of the concept and the different levels of complexity therein. For example, the issue of nondeterministic order within a chart, and between charts, is mathematically the same, but cognitive-wise, the latter is harder. However, dealing with the former first gives scaffoldings to deal with the more complex appearance. The fact that the concept appears in various ways (through different constructs – see Section 2.4) allows using it while solving different kinds of programming problems. This provides diverse opportunities for learning the concept, what is known to support transfer.

Arranging the second half of the course around projects allowed us to achieve several educational goals. First, working on projects emphasizes creation on all levels (from low- to high-level). Since ND is very prominent in LSC, this inherently involves the creation of ND, through different constructs and on various levels of complexity. It is reasonable to assume that creation activities in which ND is both inherent and significant promote the understanding of ND on the level that falls into Bloom's Creating. Indeed, we believe that a major strength of our approach is promoting the use of operative ND on this level. This is in contrast to the difficulties that were reported in using ND on this level in the context of automata theory.

Second, project-based learning promotes assembling the disconnected pieces of knowledge. As a result, the students develop a higher level of understanding, and we feel that this is another advantage of the suggested approach. It is widely believed that a high-level, unified view of the learning subject supports its retention and is a prerequisite for non-specific transfer. However, verifying this was not in the scope of the present research.

Third, there is no doubt that projects increase students' engagement and motivation. Especially, this seems to be the cae when the students can choose the project on their own, and when the language allows them to create systems that are meaningful and interesting for them (as opposed to languages that force them to spend a lot of time on low-level details). For example, two of the groups continued to work on their projects even after the course ended, although they were fully aware of the fact that this work was not going to be assessed.

Together, all these established a learning framework that seems to promote meaningful learning of operative ND. We thus believe that operative ND can be taught on the level of high-school and undergraduate programs, using LSC or some other appropriate nondeterministic programming language with similar properties.

We would like to stress two issues that arise from the findings and should be considered when applying the results of this study for developing learning materials.

First, a main advantage of our approach is that it allows the students to deal with operative ND on the level of *creating*, which is considered the highest level of learning in Bloom's taxonomy. Thus, it is very important to arrange the learning, or parts thereof, around projects and hands-on activities.

Second, in some ways, creating a nondeterministic program in LSC is easier than creating a deterministic one. As discussed in Section 3.3.2 (Creating/Multistructural), we saw a difficulty in *removing* ND and creating deterministic code. While this provides an opportunity to deal with a different aspect of the subject, it indicates a potential issue that can emerge when using a language in which ND is the default behavior.

5. Summary and Conclusions

We have studied how high-school students understand operative ND after taking a semestrial course in which they learned the nondeterministic programming language of *live sequence charts* (LSC).

Our findings show that after the course, the students demonstrated a significant understanding of operative ND, on a level that allowed them to mentally simulate nondeterministic programs on different level of complexities, and to create nondeterministic programs by themselves.

We believe that it is important to expose students to operative ND, which is inherent in nondeterministic programming languages like LSC, and in concurrent programming, and not only to the kind of ND that appears in automata.

Thus, we suggest to consider embedding the teaching of operative ND somewhere along the stream of programming courses, using a nondeterministic programming language like LSC. An interesting question for further research is the effect of learning operative ND on the learning of the kind of ND that appears in automata theory. Namely, whether learning operative ND first can improve students' achievements in the parts that deal with ND in courses on automata theory.

References

Alexandron, G., Armoni, M., Harel, D. (2011). Programming with the user in mind. *Psychology of Programming Interest Group Annual Conference (PPIG)*.

Anderson, L., Krathwohl, D., Bloom, B. (2001). *A Taxonomy for Learning, Teaching, and Assessing: a Revision of Bloom's Taxonomy of Educational Objectives*. Longman.

Armoni, M., Ben-Ari, M. (2009). The concept of nondeterminism: its development and implications for teaching. *SIGCSE Bull.*, 41(2), 141–160.

Armoni, M., Gal-Ezer, J. (2004). On the achievements of high school students studying computational models. *SIGCSE Bull.*, 36(3), 17–21.

Armoni, M., Gal-Ezer, J. (2006). Introducing nondeterminism. *Journal of Computers in Mathematics and Science Teaching*, 25(4), 325–359.

Armoni, M., Gal-Ezer, J. (2007). Non-determinism: An abstract concept in computer science studies. *Computer Science Education*, 17(4), 243–262.

Armoni, M., Lewenstein, N., Ben-Ari, M. (2008). Teaching students to think nondeterministically. *SIGCSE Bull.*, 40(1), 4–8.

Ben-Ari, M. (2010). A primer on model checking. *ACM Inroads*, 1(1), 40–47.

Biggs, J.B. (1999). *Teaching for quality learning at university*. Open University Press, Buckingham.

Biggs, J.B., Collis, K.F. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Blumenfeld, P.C., Soloway, E., Marx, R.W., Krajcik, J.S., Guzdial, M., Palincsar, A. (1991). Motivating project-based learning: Sustaining the doing, supporting the learning. *Educational Psychologist*, 26(3–4), 369–398.

Bruner, J. (1960). *The Process of Education*. Harvard University Press, Cambridge, MA.

Damm, W., Harel, D. (2001). LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1), 45–80.

Dijkstra, E.W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8), 453–457.

Dijkstra, E.W. (1976). *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, USA.

Fuller, U., Johnson, C.G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T.L., Thompson, D.M., Riedesel, C., Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *SIGCSE Bull.*, 39(4), 152–170.

Gal-Ezer, J., Beeri, C., Harel, D., Yehudai, A. (1995). A high school program in computer science. *Computer*, 28(10), 73–80.

Ginat, D. (2010). The baffling cs notions of "as-if" and "don't-care". In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE'10. New York, NY, USA. ACM, 385–389.

Harel, D. (2000). From play-in scenarios to code: an achievable dream. In: Maibaum, T., editor, *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science, 1783)*. Springer Berlin/Heidelberg, 22–34.

Harel, D., Marelly, R. (2003a). *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, New York, Inc., Secaucus, NJ, USA.

Harel, D., Marelly, R. (2003b). Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and Systems Modeling (SoSyM)*, 2(2), 82–107.

IEEE/ACM (2001). Computing Curricula 2001: Computer Science Volume – Final Report.

IEEE/ACM (2013). Computer Science Curricula 2013 (Ironman Draft).

Kolikant, Y.B.-D. (2004). Learning concurrency: evolution of students' understanding of synchronization. *Int. J. Hum.-Comput. Stud.*, 60(2), 243–268.

Lister, R., Leaney, J. (2003). Introductory programming, criterion-referencing, and bloom. *SIGCSE Bull.*, 35(1), 143–147.

Lister, R., Simon, B., Thompson, E., Whalley, J.L., Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the solo taxonomy. *SIGCSE Bull.*, 38(3).

Meerbaum-Salant, O., Armoni, M., Ben-Ari, M.M. (2010). Learning computer science concepts with scratch. In: *Proceedings of the Sixth International Workshop on Computing Education Research, ICER'10*. New York, NY, USA. ACM, 69–76.

Perrenet, J., Groote, J.F., Kaasenbrood, E. (2005). Exploring students' understanding of the concept of algorithm: levels of abstraction. *SIGCSE Bull.*, 37(3), 64–68.

Rabin, M.O., Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2), 114–125.

Schank, R., Berman, T.R., Macpherson, K. (1999). Learning by doing. In: *Instructional-Design Theories and Models: A New Paradigm of Instructional Theory*. Lawrence Erlbaum Associates, 161–181.

Schwill, A. (1994). Fundamental ideas of computer science. *Bull. European Assoc. for Theoretical Computer Science*, 53, 274–295.

G. Alexandron is currently a postdoctoral researcher at MIT, where he is using computational tools to study the behavior of students in Massive Open Online Courses (MOOCs). His research interest spans across computer, data and the learning sciences. He holds PhD in computer science education from the Weizmann Institute of Science and M.Sc. in computer science from Tel-Aviv University. Between his M.Sc. and PhD he spent few years in the industry developing programming languages and tools.

M. Armoni is a senior scientist at the Department of Science Teaching, Weizmann Institute of Science. She received her PhD from the School of Education in Tel-Aviv University, and her B.A. and M.Sc. in computer science from the Technion, Israel Institute of Technology. She has been engaged in computer science education for more than 20 years as a lecturer and a teacher, as a curricular developer, and as a researcher. She has co-authored several textbooks for high schools and for junior high schools. Her research interests are in the teaching and learning processes of computer science, specifically of various computer science fundamental ideas.

M. Gordon is a senior lecturer at the Holon Institute of Technology (HIT) in Israel. She is working in the area of natural interfaces to programming and computer science education. She has been as a research assistant at the Media Lab in MIT, where she has worked on developing the Social Robot Toolkit an ambient environment for young children to program robots. She received her PhD and M.Sc. in computer science from the Weizmann Institute of Science, working on programming languages and visual classification. She received her B.A. in Computer Science and Psychology from Tel Aviv University in Israel.

D. Harel is a Professor of Computer Science at the Weizmann Institute of Science, and served as Dean of the Faculty of Mathematics and Computer Science. He is also the Vice President of the Israel Academy of Sciences and Humanities. He has worked in logic and computability, software and systems engineering, modeling biological systems and more. He invented Statecharts and co-invented Live Sequence Charts. Among his books are "Algorithmics: The Spirit of Computing" and "Computers Ltd.: What They Really Can't Do". His awards include the ACM Karlstrom Outstanding Educator Award, the Israel Prize, the ACM Software System Award, the Eme't Prize, and five honorary degrees. He is a Fellow of ACM, IEEE and AAAS, a member of the Academia Europaea and the Israel Academy of Sciences and Humanities, and a foreign member of the US National Academy of Engineering and the American Academy of Arts and Sciences.