On Teaching Programming with Nondeterminism [*]

Giora Alexandron Michal Armoni Michal Gordon David Harel

Weizmann Institute of Science, Rehovot, 76100, Israel
{giora.alexandron, michal.armoni, michal.gordon, david.harel@weizmann.ac.il}

ABSTRACT

Non-determinism (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. The other, which we term *operative*, appears in non-deterministic programming languages and in the context of concurrent and distributed systems. We believe that it is important to teach the two types of ND, especially as ND has become a very prominent characteristic of computerized systems. Currently, students are mainly introduced to ND of the first type, which is known to be hard to teach and learn. Our findings suggest that learning operative ND might be easier, and that students can reach a significant understanding of this concept when it is introduced in the context of a programming course that deals with a non-deterministic programming language like the language of Live Sequence Charts (LSC). Based on that, we suggest teaching operative ND in the context of concurrent and distributed programming, a topic which is covered by a new knowledge area that was added in Computer Science Curricula 2013.

Categories and Subject Descriptors

K.3.2 [**COMPUTERS AND EDUCATION**]: Computer and Information Science Education—*Computer science education*

General Terms

Languages

Keywords

Non-determinism, constructionism

1. INTRODUCTION

The concept of non-determinism (ND) was first introduced to computer science (CS) by Rabin and Scott in their work on non-deterministic finite automata [19], for which they were given the Turing Award. In Schwill's work on fundamental ideas of CS [20], it is listed under the category of *programming concepts*. As a fundamental idea, ND appears in various domains and contexts. In CC2001 [15] it is covered mainly in the elective unit on Automata theory (AL7). At the time that curriculum was developed, ND was a much less prominent characteristic of computerized systems. This has changed, as reflected, for example, in a new knowledge area on parallel and distributed computing that was added in CC2013 [16]. The rationale underlying 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 is one of these essential concepts.

Some researchers view different appearances of ND as manifestations of the same thing (this viewpoint is expressed for example in [2, 9]), but they can also be seen as different types of ND. We distinguish between two different types of ND. The first is the one that appears in the context of automata theory and formal languages (through non-deterministic automata). On this type of ND there is a predetermined criterion for success, 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 non-deterministic programming languages. Examples of such languages include Dijkstra's *guarded commands* [8]; *logic programming* languages, such as Prolog, usually have a non-deterministic 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) [7] (which is mainly a language for reactive systems development), also have a non-deterministic 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.

In practice, it seems that students are usually introduced to ND of the first type, in the context of automata theory and formal languages [3]. This type of ND is known to be difficult to teach and learn [2, 4, 5]. With respect to the learning of operative ND, Ben-David Kolikant [17] identifies this concept as the main source of difficulty in learning concurrent programming.

We believe that it is important to introduce students to both types of ND. Thus, we suggest to embed the teaching of operative ND in a course that involves concurrent and asynchronous programming with non-deterministic programming languages like LSC. The advantage of this approach is that it does not require adding an isolated topic to the curriculum. It can be implemented by extending a programming course, which stands on its own merit, with a

[*]This research was partially supported by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013). The work of the first author was supported by a grant from the Azrieli Foundation.

new context. This suggestion is based upon our experience with teaching LSC to 12th grade high school students, and upon the results of the study conducted on par with the teaching process. The results show that after the course, the students showed a significant understanding of operative ND, on a level that allowed them to create and execute programs that included ND in various scopes and degrees of complexity.

2. LIVE SEQUENCE CHARTS

In this section we briefly describe the language of Live-Sequence Charts (LSC) and its development environment, Play-Engine. The language was originally introduced in [7] and was extended significantly in [12] and [13]. LSC is a visual specification language for reactive system development. LSC and 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 Figure 1.

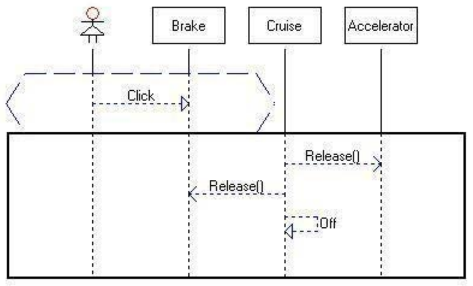

Figure 1: LSC chart

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 it self off.

2.2 The play-in/play-out methods

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, which is implemented in Play-Engine. With play-in, the user specifies the scenarios in a way that is close to how a real interaction with the system occurs, by *playing* with the GUI. For more details, see [11, 12, 13].

The play-out method (originally introduced in [13]), also implemented in Play-Engine, is the underlying execution engine that makes LSC (which is a declarative programming language) directly executable/simulatable. For more details see [12].

2.3 Non-determinism in LSC

LSC is a non-deterministic 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 various abstraction mechanisms that allow to describe aspects of the system behavior without being forced to introduce determinism into the implementation, when it is not derived from the requirements. Due to lack of space, we do not review these mechanisms here. As a model of computation, LSC also includes parts that implement an asynchronous, thus non-deterministic, semantics.

3. THE STUDY

Our research question was how high-school students understand operative ND after learning the non-deterministic language of LSC. The context of the study was a forty-five hours semestrial course given to nineteen 12th grade (age:17-18) high-school students majoring in CS. Students' experience in CS included two introductory computing courses, a course on computer organization and assembly languages, and a course on computational models (in order to make space for the LSC course, this was a shortened version, which did not include non-deterministic models). In none of these courses they were introduced to the concept of ND. The LSC course was a mandatory course and was developed and executed as a pilot course aimed at teaching scenario-based programming and reactive systems development with LSC. The course structure was arranged according to the "zipper principle" [10], with theoretical lectures followed by hands-on experience in the lab. The last few lessons were devoted for developing final projects in groups. 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 write, comprehend and modify LSC programs. The data that we collected included the projects, exam questions, class notes, and post-interviews held with representative students.

3.1 Assessing students' understanding

To assess students' learning, we used a two-dimensional taxonomy. Each category in the taxonomy represented a certain level of learning. By analyzing the data, we estimated the extent to which each level of learning was achieved. The two-dimensional taxonomy is built upon two existing taxonomies. The vertical axis is based upon Bloom's taxonomy in its revised form [1], and the horizontal axis is based upon the SOLO taxonomy [6]. From Bloom's taxonomy we chose to concentrate on one intermediate category (Applying) and one higher category (Creating). From the SOLO taxonomy we chose to focus on the three intermediate categories (out of five): Unistructural, Multistructural, and Relational. The operationalization of the categories with respect to LSC is given on the next section. This scheme is a variation of the combined taxonomy suggested by Meerbaum-Salant et al. [18] for assessing students' learning of CS concepts. The taxonomy is presented in table 1, together with the type of analysis that was used on each category.

3.1.1 *Operationalization*

We follow the interpretation suggested in [18], 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

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

Table 1: The taxonomy and the analysis conducted on each category

the context of LSC, the ability to track and simulate pieces of code that contain a non-deterministic 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 non-deterministic element/s.
From the SOLO taxonomy:
* Unistructural: A local perspective. The interpretation to LSC means acting in the scope of a single chart.
* Multistructural: A perspective that incorporate multiple LSC charts.
* Relational: A holistic perspective, referring the whole program or a property of it.

For example, the interpretation of Creating/Multistructural is creating a program that contains ND which involves multiple charts.

3.2 Findings

Below we describe a subset of the findings, and we intend to give a complete report elsewhere. From the quantitative analysis, we present a table summarizing the quantitative results, and exemplify the analysis with the category of Applying/Unistructural. From the qualitative analysis, we present the findings related to category of Creating/Multistructural.

3.2.1 Quantitative findings

The quantitative analysis was based on the exam questions, and was conducted as follows. For each category that was quantitatively assessed, we first scanned the relevant questions in the exams, and mapped into this category the questions that required the level of understanding that this category measures (overall, we used four questions. Two for the category of Applying/Unistructural, and one for each of the other two categories that were quantitatively assessed). Then, students' answers to these questions were graded, with the grades referring only to the non-deterministic element of the question. Finally, the category was given the average score of the answers belonging to it.

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

The quantitative findings are exemplified by the analysis of the category of Applying/Unistructural, which is shown below. Table 2 summarizes the quantitative results.

Applying/Unistructural. The question contained a chart describing some scenario (taken from the specification of a calculator). The chart is shown in Figure 2. 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 subquestions got the same weight. This question is classified as Applying because it requires the students to mentally simulate a given algorithm, and as Unistructural because this algorithm resides in the scope of a single chart.

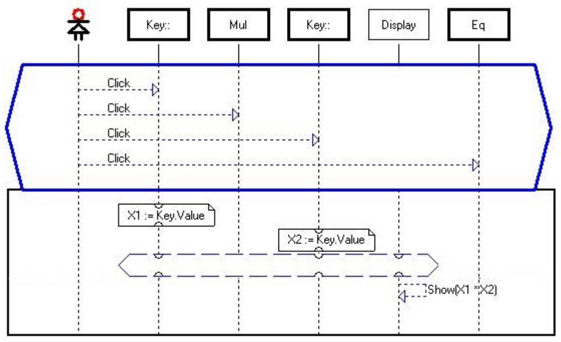

Figure 2: Applying/Unistructural

Summary of the quantitative results. Table 2 summarizes the score calculated for each category. N stands for the number of answers considered. Since some of the questions that we used were elective, each questions was answered by a different subset of the students (this might induce some bias into the results, but since we are interested in the trend, it was more important for us to include as much data as possible). As can be seen, students' achievement on these categories are satisfactory.

	Unistructural	Multistructural	Relational
Applying	83%, N=26	76%, N = 18	
Creating	100%, N=10		

Table 2: The quantitative results

3.2.2 Qualitative findings

This analysis was based on students' final projects, on post-interviews held with four representative students (one student per group), and on class notes. The students were chosen more or less at random, based on their availability. The qualitative analysis was validated with an expert who was not part of the research team. The expert was requested to analyze in depth the data concerning one of the projects, map it into 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 different arguments. Overall, the process reinforced our confidence in the findings, and enriched them with another perspective. Below we exemplify the qualitative analysis with the results of the analysis of the category of Creating/Multistructural.

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. This group 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. For example, 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 non-deterministic. The order of execution can vary 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=Interviewer, S=Student):

“ 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 allows them to ignore unnecessary implementation details such as the order of execution. This seems to reduce the cognitive load involved in programming.

However, we also saw evidence of difficulties in dealing with ND on this level. For example, student #2 showed that he was able of mentally simulating ND in the context of several charts (this example belongs to Applying/Multistructural, which is not shown here), but he then mentioned that actually only one of these non-deterministic behaviors was desired, and that he was not sure how to discard the other ones. So, this is actually a difficulty in *removing* ND and creating a deterministic code, and this is a potential issue that can emerge when using a language in which ND is the default behavior.

3.2.3 Summary of findings

To summarize the findings, we saw that:
i) Students were able of understanding non-deterministic 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 non-deterministic element of the system. On this level we saw problems when the ND stemmed from the interleaving of multiple charts.
ii). With regarding to the ability of creating non-deterministic systems, we found that:
a. Almost all the students were able of creating 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 this level when it was associated with a high-level of concurrency.

4. DISCUSSION AND CONCLUSIONS

Teaching operative ND in the context of a programming course that includes hands-on experience of building and executing systems, is inline with the ideas of *constructionism* [14]. As operative ND appears in LSC in different ways, using this language provides diverse opportunities for learning the concept. Furthermore, since LSC is non-deterministic in its nature, using it creates a learning environment in which ND is the "normal situation", as suggested by Dijkstra ([9], p. xv). Together, these help to construct a pedagogical framework that seems to promote the learning of operative ND.

Following this, and due to the increased importance of ND in current computing, we believe that operative ND can and should be taught on the level of high-school and pregraduate programs. As noted, this does not require opening the curriculum, but can be done by adding a new context to a course on concurrent and asynchronous programming.

5. ACKNOWLEDGMENTS

The authors would like to thank Nir Eitan, Ronit Ben-Bassat Levy, Zehava Levin and Rivka Taub for their help in conducting this research.

6. REFERENCES

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

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

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

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

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

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

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

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

[9] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1976.

[10] J. Gal-Ezer, C. Beeri, D. Harel, and A. Yehudai. A High School Program in Computer Science. *Computer*, 28(10):73–80, Oct. 1995.

[11] D. Harel. From Play-In Scenarios to Code: An Achievable Dream. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 22–34. Springer Berlin / Heidelberg, 2000.

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

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

[14] I. E. Harel and S. E. Papert. *Constructionism.* Ablex, Westport, CT, USA, 1991.

[15] IEEE/ACM. Computing Curricula 2001: Computer Science Volume – Final Report, 2001.

[16] IEEE/ACM. Computer Science Curricula 2013 (Ironman Draft), 2013.

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

[18] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari. Learning computer science concepts with scratch. In *Proceedings of the Sixth international workshop on Computing education research*, ICER '10, pages 69–76, New York, NY, USA, 2010. ACM.

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

[20] A. Schwill. Fundamental Ideas of Computer Science. *Bull. European Assoc. for Theoretical Computer Science*, 53:274–295, 1994.