The Effect of Previous Programming Experience on the Learning of Scenario-Based Programming [*]

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

In this paper we present qualitative findings on the influence of previous programming experience on the learning of the visual, scenario-based programming language of *live sequence charts* (LSC). Our findings suggest that previous programming experience leads programmers not only to misunderstand or misinterpret concepts that are new to them, but that it can also lead them to actively distort the new concepts in a way that enables them to use familiar programming patterns, rather than exploiting the new ones to good effect. Eventually, this leads to poor usage of some of the new concepts, and also to the creation of programs that behaved differently from what the programmers expected. We also show that previous programming experience can affect programmers' attitude towards new programming concepts. Attitude is known to have an effect on performance. Since LSC and its underlying concepts are of growing popularity in the software engineering community, it is interesting to investigate its learning process. Furthermore, we believe that our findings can shed light on some of the ways by which previous programming experience influences the learning of new programming concepts and paradigms.

Keywords

Scenario-based programming, Visual Languages, Constructivism

1. INTRODUCTION

The idea that prior experience significantly influences the way one acquires and uses new knowledge is the basis of the learning theory of *constructivism.* According to constructivism, learning is not a passive process, in which knowledge is received, but an active and subjective process, in which new knowledge is built upon existing constructs of knowledge and extends them (for more on constructivism in education, see [23]). In science education research, the constructivist approach has become very influential; for example, in the study of misconceptions (see the work of Smith et al. [22], and others). The application of constructivism to computer science education (CSE) is relatively new, but has become very influential since then, as indicated by the number of references to Ben-Ari's paper on the subject [4].

The *schema theory of learning* [2] also implies that experience plays a significant role in the learning process. This theory describes knowledge as arranged in a hierarchical structure around cognitive constructs called *schemas.* A schema encapsulates pieces of knowledge around a specific functionality. When this functionality is required – for example, when one is solving a problem, the relevant schema is retrieved and applied. Rist describes programming knowledge as arranged in plan schemas [19]. A plan schema is "an abstract solution to a common programming problem; It stores a plan to achieve a common goal" (pg. 175). Plan schemas are developed while solving programming problems in the context of specific programming languages. Thus, it is reasonable to believe that different languages and paradigms will lead to the creation of different programming schemas, and that this will significantly affect the learning and usage of new programming languages. About thirty years ago, a question in this spirit was raised by Wexelblat, in a paper titled "The consequences of one's first programming language" [25].

The subject of the first programming paradigm, and the language through which it should be introduced (hereafter, we use the term 'first language', but we actually mean the language and the main paradigm that underlies it), is a principle question in CSE. This question is the core of a storming debate, that was humoristically described by Gal-Ezer and Harel as a 'culture war' [9]. A recent and comprehensive survey on the subject of the first language can be found in [24].

Reviewing the body of research on this subject yields that arguments that base the importance of the first language on its effect on future learning of other languages are used quite often, but usually these arguments are not empirically based. As far as we know, studies on the role of previous experience in the learning of a *new* language and paradigm are quite uncommon. Empirical studies that look on the role of experience in programming usually concentrate on

[*]This research was partially supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

the differences between novices and experts, and study these differences in the context of a programming language that is *known* to the experts. See for example the works of Adelson and Soloway [1], Burkhardt et al. [5], Détienne [7], and others. We study the role of experience in the learning of a programming language and paradigm that are *new* to the experts. In this area, we are familiar only with the work of Sharp and Griffyth [21]. They analyzed the correlation between programmers' experience and programmers' understanding of object-oriented concepts. However, their analysis was based on a survey, which means that it relied on students' subjective perception of their level of understanding of the course topics. Relying on self assessment for measuring understanding in known to be unreliable, as there are documented gaps between understanding and sense of understanding. Also, due to the nature of collected data, a survey is more appropriate for verifying existing hypotheses. Since our goal is to obtain new insights and a deeper understanding of the phenomena, we chose to use a qualitative analysis of in-depth interviews that were based on students' projects.

When planning a study on the influence of programming experience on the learning of new programming concepts, it is important to use a programming language that the programmers are not familiar with, not only with respect to the syntax, but also with respect to the underlying concepts. LSC ([6], described in section 2), which introduces a new, inter-object paradigm termed scenario-based programming, and uses visual syntax, provides such an opportunity. In this paper we present findings from such a study. By analyzing several phenomena observed in the programming behavior of nine CS graduate students, we show how previous programming experience led the students to misinterpret and distort some of the new concepts that LSC introduces, and consequently to write erroneous programs.

The rest of the paper is organized as follows. In section 2 we present the principle concepts of 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 by Damm and Harel in [6], and was extended significantly in [14] and [15]. LSC is a visual specification language for reactive system development. LSC and the Play-Engine are based on three main concepts, that 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, as seen by the user, and may include possible, necessary or forbidden actions. For example, cash withdrawal is a basic functionality of an ATM machine. In LSC, it can be captured in a scenario that describes the system behavior in cash withdrawal, and will include the interactions between the withdrawer and the system, and between the internal parts of the system. Since a scenario usually involves multiple objects – "one story for all relevant objects" ([14], pg. 4) – scenario-based programming is *inter-object* by nature. Returning to the ATM, a scenario-based specification of an ATM will describe the ATM as a collection of such user-view inter-object scenarios. Scenarios in LSC are also *multi-modal* in nature. That is, LSC events have behavioral properties that define modalities of behavior in three dimensions: must vs. may, allow vs. forbid, and execute vs. observe.

Syntactically, a scenario is implemented in a live sequence chart (see Figure 1). The chart is composed of two parts – the *prechart*, and the *main chart*. The prechart is the upper dashed-line hexagon, and it is the activation condition of the chart. In case that the events in the prechart 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 Figure 1 is taken from a sample implementation of a *cruise control*. It describes a scenario of stopping the cruise control when the foot brake is pressed. When this happens, the cruise unit releases control of the brakes and the accelerator, and turns itself off.

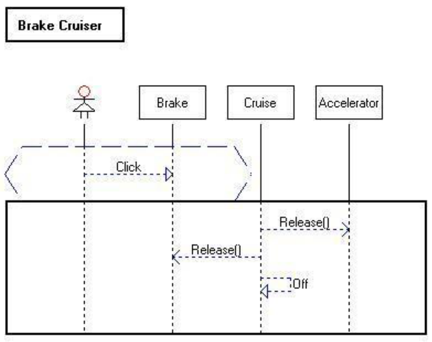

Figure 1: A simple LSC chart.

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 [12, 14, 15], which is implemented in the Play-Engine, LSC's development environment. With play-in, the user specifies the scenarios in a way that is close to how real interaction with the system occurs. For example, to implement the scenario described in Figure 1, the user clicks the brake (represented by a button in the GUI model of the cruise control), and then manipulates the brake, accelerator, and cruise unit, to obtain the actions in the main chart. Using this direct interface programming method, users who are not familiar with LSC (or even with other programming languages) can program the behavior of an artifact relatively easily.

2.3 The play-out method

LSC has an operational semantics that is implemented by the play-out method (originally introduced in [15]). 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 fly. For more details see [14].

3. THE STUDY

In this section we present an empirical study conducted as the first part of a larger research effort, in which we investigate educational issues involved in the learning of LSC and scenario-based programming. Our objective in this phase is to get a deeper understanding of the research subject. Thus, our methodology is purely qualitative and follows the ideas of *grounded theory* [10], in the sense that we do not obligate ourselves to look at specific phenomena, but collect rich data, from which interesting phenomena can emerge.

3.1 The research question

The research question that is the focus of this paper is in what ways programming experience that is mainly procedural, object-oriented, and sequential, affects the learning and use of scenario-based decomposition and other abstraction mechanisms of LSC.

3.2 Research setting

The setting of the study was based on the course "Executable Visual Languages for System Development", given by the fourth-listed author in the Fall term of 2010-2011 at the Weizmann Institute of Science [1]. This course presented various aspects of reactive system development, and concentrated on two approaches to the specification, design and implementation of systems: the intra-object approach (through Statecharts) and the inter-object approach (through LSC). About half of the course was devoted to Statecharts and the intra-object approach, and about half to LSC and the inter-object approach.

Course assignments included an implementation project to be carried out using LSC. The students were directed to choose a system (of reasonable complexity) that they find appropriate, and implement it in LSC. Student projects included, among other things, modeling the blood's glucose level control system, modeling animal behavior, and modeling a variety of electronic devices. With some variations, this course is given for the third time. A report on the first experience of teaching the course can be found in [13].

3.2.1 Research population

The student population was composed of graduate students studying towards an MSc or PhD in computer science. Nine students taking the course participated in this study. Among these, six were CS graduates, two majored in Biology/Bioinformatics, and one majored in Electrical Engineering (EE).

Previous programming experience.

We characterize students' previous experience with respect to procedural and object-oriented programming. In the following sections we refer to this characterization when analyzing students behavior in the light of their previous programming experience. All of our students learned procedural programming on early stages of their programming studies – through Pascal (in high school), C, or C++ (learned as a procedural language, without getting into object-oriented programming). All of the six CS graduates also took advanced courses in object-oriented programming (OOP), and had a few years of programming experience in industry, mainly with C++ and Java. We define the group of the CS graduates as group A, and refer to their experience as type A experience. The three non-CS graduates had less significant programming experience (in general, and specifically with OOP). The EE graduate was familiar with procedural programming in C. From the two biology graduates, one took two C++ courses in university – one introductory course using the language as a procedural language, and a more advanced course teaching object-oriented programming. The other's experience was mainly with scripting languages. We define the group of the non-CS graduates as group B, and refer to their more limited experience as type B experience.

[1]http://www.wisdom.weizmann.ac.il/∼michalk/VisLang2011/

3.2.2 Data collection tools

i) Pre-interviews: These were mainly used to characterize the student's previous programming experience. The students were also given a programming task, which is less relevant in the context of this paper.
ii) Learning styles questionnaires: These were used to characterize students' learning preferences, in order to see if there are connections between the individual preferences and the learning process.
iii) Student projects: The projects that the students implemented in LSC.
iv) Post interviews: In the interviews we asked each student about the LSC project. The post interviews were semi-structured and were divided into two parts. The first included questions such as: "Why did you choose this specific project?", "Why did you take these specific design decisions?", and also questions on the semantics of the language constructs. Regarding each topic, we used follow-up open questions when we felt that more interesting information could be revealed, or to verify that our interpretation of the answer was correct. The second part, which is less relevant in the context of this paper, included solving a programming task.

3.3 Findings

From our data, a number of interesting phenomena emerged. We focus here on a subset thereof, and we intend to report on the remaining phenomena in a future publication.

3.3.1 Perception of LSC as a programming language

In several cases and situations, some students expressed their perception that working with LSC is 'not really programming'. This seemed to be related to the gap between the lower abstraction level of the languages with which these students were experienced, and the higher abstraction level of LSC.

Among these cases, we concentrate on a somewhat extreme example that included also a negative attitude, which was observed in the interview held with student #2. The main experience of this student in his graduate studies was with Java. He then worked few years in industry, mainly developing drivers in C. His professional experience also included writing HDL code in Verilog.

The student expressed dissatisfaction with the fact that the language did not allow accessing low level details:

> “S: What you can do with LSC is very limited because LSC does not elevate everything up [...]. You can’t make computational things. You can’t interact with the computer [...]. When you program [with a ‘real’ language], you have access to the memory, you can play with bits, etc.”.

The student wanted control:

> “S: I am in favor of giving power to the creator [i.e., the programmer]”.

He felt that LSC does not give him this level of control. Obviously, higher abstraction means less control on low-level details. However, the freedom obtained by the higher abstraction was interpreted by the student as a negative thing:

> “S: It is not good to have degrees of freedom in your system. You shouldn’t have them”.

These claims were raised when questioning the student on his final project, in which he modeled the behavior of a house pet. This project actually did not require any low level operations of the kind that the student was missing. Still, he looked for these operations, and was uncomfortable with the fact that they were not natively supported by the language.

Moreover, it seemed that the absence of control led to a negative attitude, which eventually affected his performance. In some way, the student interpreted the degrees of freedom as permission to be inaccurate. In a few cases, the student mentioned that he didn’t bother too much about being accurate in modeling the pet’s behavior. He said that anyway the system will simulate some behavior, and since animals are complex, this simulation could be referred to (in retrospect) as a natural behavior:

> “S: [...] even if there is a bug, let’s say the delivery arrives and the cat is sleeping and doesn’t run away, I can say yes, it didn’t hear it, so it continues to sleep”.

The student’s concluding remark was:

> “S: This is programming for people who do not want to know programming”.

For this student, accessing the memory, manipulating bits, and in general doing low level operations, are the most fundamental aspects of programming. Thus, knowing how to program means knowing to use these operations. Since LCS does not have these kinds of operations, the student perceived it as a language that can be used without knowing how to program.

3.3.2 Procedural and sequential design

The second phenomena found was the echo of students’ experience with procedural programming in their patterns of design in LSC.

The basic idea behind procedural programming is to decompose the programming task into a series of procedures (aka functions or subroutines). Modularity is achieved by breaking long procedures into smaller ones, encapsulated in autonomous pieces of code that can be reused. This allows arranging the program in a hierarchical structure that reflects the abstraction layers. The ideas of procedural programming also reside in the object-oriented paradigm and languages that support it (such as C++ or Java). In the context of this section, by procedural programming we refer not only to the paradigm itself, but also to the procedural components that reside in languages that support object-oriented programming (OOP).

There are a few differences between the execution model of LSC and the standard execution model of procedural programming. We focus here on two major differences:

Sequential vs. concurrent execution: The execution of a sequential procedural language proceeds one step at a time. At each point of the execution, there is no choice regarding which step to take next, and only one subroutine progresses. The caller subroutine (which was called by another subroutine, and so forth) does not progress until the callee finishes. LSC’s execution model is totally different: All the active charts can progress simultaneously. Once a chart is activated by another chart, both charts are independent and none of them needs to wait for the other (unless they have inter-dependencies. See below).

Unification of events: In procedural languages, actions are executed in a local scope. This means, among other things, that if a function has multiple active copies (e.g., in a recursive call), actions that refer to the same source item in the code are independent – they are different entities in the ‘world’. LSC semantics is very different. LSC is a declarative language, and uses the concept of unification. Unification is a principle concept of declarative languages and logic programming: Essentially, unification means that identical actions that appear in different charts, or in different active copies of the same chart, are considered the same action. Entities in the code are only references to the real entities in the ‘world’ (for more details, see [14]). At run-time, identical actions are constantly unified in all the active charts, and executed together. It is a single action/execution, and all the active charts that contain this action progress simultaneously. Also, the semantics of LSC requires that an action can be executed only if all its occurrences can be executed. Very roughly, this means that this action is (one of) the next action in all the charts in which occurrences of this action appear. So, as opposed to the local scope of the actions in procedural languages, actions in LSC are considered within a scope that contains all the active charts, and this creates inter-dependencies between the charts.

One recurring pattern found in students’ projects was the imitation of functions using dedicated charts. By that, we mean that students tended to break behavioral scenarios into smaller functional pieces, encapsulate these pieces in dedicated charts, and call these charts from ‘higher level’ charts. We refer to this as the *function pattern*. Indeed, some students actually called these charts ‘functions’, and their explanation for this structuring were code reuse, modularity, etc. This structuring, and the motivation behind it, are the essence of procedural programming, which all of our students were experienced with.

The function pattern is not the natural kind of decomposition encouraged by LSC. Scenario-based programming with LSC encourages decomposing the system into behavioral and stand-alone pieces, encapsulated in charts that are activated as a response to system/user events. Students’ function charts had very different characteristics. We define a chart as a function chart if it meets all the following conditions: i) the chart does not represent autonomous behavior of the system. ii) the chart is activated by another chart (and not as a response to system/user events); iii) the activation condition of the chart contains a single dedicated event, that is used only for this purpose.

From seven projects (some students worked in pairs), five projects contained this pattern. Here we did not see any difference between group A and B. Indeed the students in both groups had procedural experience. We note that this pattern is not necessarily bad – code reuse and modularity are desirable objectives. However, using it without considering the operational consequences of the differences between LSC execution model, and the sequential procedural execution model from which this pattern is borrowed, can lead to an erroneous behavior. The programming behavior of using functions is the result of previous experience, as it has not been taught or discussed as part of the LSC language.

A typical example can be seen in Figure 2, taken from the project of student #3. The left chart is the caller (only a piece of the chart, which contains the relevant actions, is shown). The two charts on the right are the callees – the function charts. The event 'incRPM(7)' in the left chart activates the upper right chart, and the following event, 'Log()', activates the lower right chart (the events and the activated charts are marked with arrows).

The student used a top-down design strategy and built a procedural structure that goes from higher to lower abstraction levels. The caller chart represents a higher level flow, that calls two sub-routines. In the abstraction level of the caller, the student perceived the two calls as atomic operations, black-boxes, and intended that the second will happen only after the first is finished. However, because the charts can progress simultaneously, there is no guarantee that this will actually happen. The caller chart can emit the event 'incRPM', and then emit the event 'Log' immediately after. As a result, the two function charts will be opened, and from this point there is no necessary order between them. This interpretation was verified with the student.

So, the student encapsulated a low level computation in a function chart, and called this chart from another chart that describes a higher level flow. The student perceived the function call as an atomic operation (as happens in a procedural execution), but due to the simultaneous execution, this led to an erroneous behavior.

Some of the students, including student #3, did identify the potential problem of simultaneous execution. However, their solution was quite surprising – they chose a somewhat Gordian knot (aka Alexandrian) solution, and bypassed the problem by imposing a sequential execution over the run. This pattern, which we will call the *sequentialization pattern*, is exemplified in Figure 3 (it is also taken from the project of student #3). The figure contains two charts. The left chart is the caller, and the calling event ('increment()') is marked with an ellipse. The right chart is the function chart, invoked when the event 'increment()'(in its prechart, marked with an ellipse) is emitted. The student used the unification rules described above. After calling to the right chart, the caller chart must wait for the event operationDone() that the callee chart emits when it finishes, in order to progress. This design imposes a sequential procedural-like execution. Sequentializing the run in order to bypass the problems raised by simultaneous execution was also seen in the projects of three other students, and in the projects

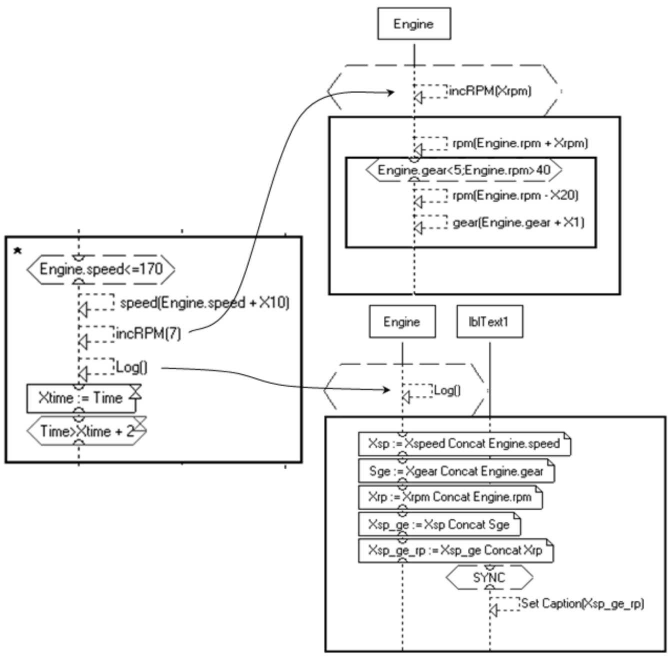

Figure 2: Imitating a function call.

of students taking this course in the past. This is a poor use of LSC – the students did not fully exploit the richness and novelty of the language, and embedded a sequential program into the structure of a concurrent language.

In some sense, this behavior is reminiscent of the behavior seen in the 'Perception' pattern (section 3.3.1). The previous experience not only led the students to misunderstand or misinterpret the new concepts, but it also led them to actively look for the known concepts and ways to imitate them in the new language.

3.3.3 Interpretation of the new abstraction concept of symbolic instances

One of the concepts introduced by LSC is that of *symbolic instances*. This is a principle abstraction concept of LSC, which enables defining general behavior for instance/s of a certain class. The concept of symbolic instances relies on the existence of class:instance relationships. Like classes, symbolic instances enable defining behavior for multiple instances of a specific class. However, symbolic instances extend the expressivity power of class:instance relationships. The symbolic instance includes a binding expression and has the *universal*/*existential* modality. The binding expression is evaluated at run-time for all the class objects, and the ones that match it are bound according to the following rule: If the symbolic instance is universal, all of them are bound, and if it is existential, only one of them is bound. Then, the behavior that was defined for the symbolic instance is applied to the bound object/s. An example is given in Figure 4. It is taken from the project of student #4, who modeled a traffic junction. It defines the behavior of a pedestrian

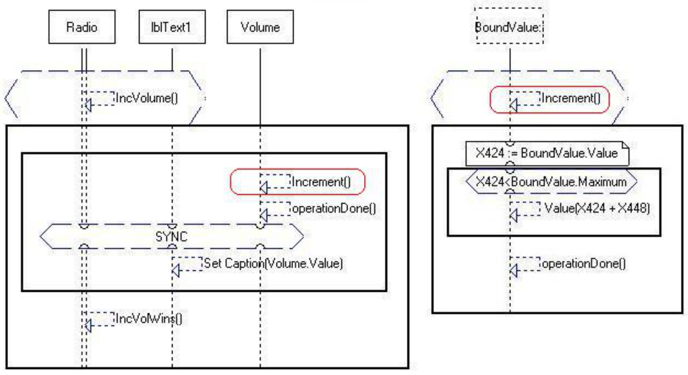

Figure 3: Forcing a sequential execution.

who is waiting to cross the road. Once the pedestrian clicks on the crossing button, the walkers traffic light sends a message to the cars traffic light to turn red (in another chart, not shown here, the cars traffic light then sends a message to the pedestrians traffic light to become green). The pedestrian is represented by the the symbolic object 'Walker', which is an *existential* symbolic instance (indicated by a dashed frame), and has the binding expression 'Wait=True'. This means that this behavior will be applied to one of the pedestrians who are waiting to cross the road. The rationale behind using an existential symbolic instance in this case is that it is enough that one pedestrian will push the button, and it does not matter which pedestrian.

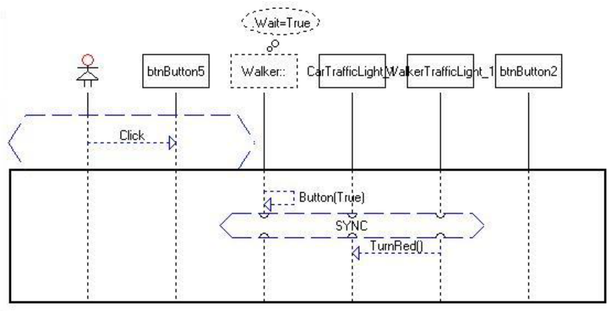

Figure 4: Symbolic instances.

The extended power of symbolic instances over classes is expressed by: i) the binding expression allows applying the behavior only to the specific objects that match the binding rule. ii) the universal/existential modality allows applying this behavior to all or one of the objects. iii) the scenario-based decomposition allows defining the behavior with respect to a specific scenario, according to the chart in which the symbolic instance is defined. Interestingly, students from group A (the group of students which had a significant programming experience with OOP) seemed to have problems with using the first two:

The binding expression: The binding expression 'chooses' dynamically a subset of objects (the objects that match the expression). In case the expression is trivial (i.e., 'id >= 0', where all the id's have non-negative values), it always chooses all the objects of the class. We call this a *meaningless* expression. We consider a binding expression as *meaningful* if it uses an expression that relies on a dynamic property that changes during the run, and chooses only a subset of the objects. Three out of nine students did not use meaningful expressions at all. All of these students were from group A. One of them indeed said that he did not understand the semantics of symbolic instances. The other two were not asked about it (since the phenomena emerged from the data, we analyzed it after the interview). In group B, all the students used meaningful binding expressions.

Universal vs. existential: This modality also seemed to be confusing for students with type A experience. While all the students in group B used it in both ways, and backed it up with a rational explanation, students in group A had some difficulties understanding it. For example, student #3 said:

> "S: The truth is that I don't really remember the difference [between universal/existential]. [...] We didn't consider the option of *universal*, the *existential* was the default for us".

Student #5 was asked "And what was the difference if this item was made *universal*?", and answered:

> "S: Ah, maybe that's the way it went, and it worked. It's hard for me, maybe I didn't... [the student feels uncomfortable]".

Another indication for this misunderstanding was given by an odd use pattern seen only in the projects of students from group A. The students used a symbolic instance that always chooses exactly one object (usually by using an implicit binding; not describe above), and marked it as *existential*. However, the *existential* has no special meaning in this case. Existential means choosing one of the matching objects, but if only one object matches, then there is no choice. We note that this is not an error in itself, but it seems that none of the students who had this pattern in their code noted this delicate semantic issue. Two of them actually said that they did not notice it. Their explanations led us to think that they used *existential* in these cases because they think that existential means 'one', and missed the idea of 'one of..'.

4. DISCUSSION

Our conceptual framework is based upon constructivism, and the schema theory of learning. Combining these two theories implies that learners will interpret new programming concepts through the prism of their previous programming knowledge. This knowledge is arranged in schemas that encapsulate abstract solutions to programming problems. When solving a problem using a new programming language, learners will try to apply their existing schemas. Here we concentrate on specific effects of applying schemas that were acquired in the context of procedural and object-oriented programming.

4.1 The effect of working on a low abstraction level

Working on a high abstraction level means concentrating more on *what* the program does, and less on *how* it does it. A straightforward implication is that the programmer has less control on the execution of operations that lie on an abstraction level that is below the one on which the programmer works.

In section 3.3.1 we presented findings related to how some of the students perceived the high abstraction level of LSC. We showed that some felt that the high abstraction level does not give them enough control, though the goals of their program were achieved without getting into lower level details. We believe that this subjective feeling is strongly related to our students' previous programming experience. Since the students were used to working on lower abstraction levels, it determined their perception of what the 'right abstraction level' is. When moving to a higher abstraction level, they could not control things that they used to control before, and thus felt that they lose power.

For example, student #2 mentioned the kind of operations that he missed in LSC. These exist in the languages that he was familiar with. On the other hand, he did not mention operations that were *not* supported by the languages he was familiar with. Thus, it is more likely that his expectations were the result of his experience, rather than the result of a basic observation about what programming languages should include.

Learning abstraction is known to be difficult, in general and in the context of computer science and software engineering. In studies that deal with abstraction in the context of programming, the setting is usually a specific programming language, used on different abstraction levels. See for example the work of Haberman [11] (and references thereof). Also, it is believed that abstract thinking evolves with expertise in programming, and that this is one of the abilities that distinguish novices from experts (see [11] and others). Our findings suggest that the difficulties involved in moving between abstraction levels exist also when moving from a lower level programming language to a higher level one, and that in this case increasing the abstraction level causes difficulties for experts.

With LSC, it might be that the dissonance between the familiar abstraction level and the higher abstraction level of LSC was strengthened by the fact that LSC is a visual programming language. According to Petre [17], visual languages usually describe information on a higher abstraction level and are viewed as less formal than textual languages.

For example, student #2 was convinced that the visual notations are actually translated into textual code (they are not; in the Play-Engine, the visual notations are executed directly). When the interviewer asked him "and did it bother you that there is no [textual] code here?", he answered: "Yes. Well, there is code, but it is hidden". According to Hazzan [16], learners cope with unfamiliar concepts by reducing the abstraction level of the new concepts, in order to make them more concrete and familiar. Through this interpretation, the student's assumption that there is textual code beneath the visual code can be seen as an attempt to lower the abstraction level in his perception. In this case, assuming that there is hidden textual code makes things more concrete for him in two ways – first, it reduces the abstraction level of the notations; second, the textual notations are more familiar, and are thus more concrete for the student.

Of course, individual characteristics can also affect one's programming behavior. Felder and Silverman suggest *Learning styles* [8] as a framework for classifying learners according to their individual learning preferences. Since our students filled a learning styles questionnaire at the beginning of the course, we checked out student's #2 questionnaire. It turned out that he had some preference for the 'sensory' and the 'visual' poles (7 in each). However, these preferences are considered moderate according to the questionnaire's scale. Also, Felder suggested that most engineering students are visual and sensory. Hence, it seems that learning styles do not supply a significant alternative explanation here.

This observed phenomenon seems relevant not only to the narrow context of LSC and other visual languages, but also to other contexts in which one learns a higher level language after lengthy experience with a lower level one. For example, we could probably hear most of the above quotes from someone learning Java (as the high-level language) after C (as the low-level language). Our findings indicate an attitude of resistance rooted in the change of abstraction level in such cases.

Attitudes play an important role in problem solving performance, both in general (in the work of Schoenfeld [20] it is categorized under *beliefs*), and in the context of programming and other computer science oriented tasks. For example, Armoni et al. [3] reported that students who perceive reduction as a non-legitimate problem solving tool show lower performance in using it. Thus, we believe that these findings will be of interest to researchers and practitioners involved in computer science education in its wider context.

4.2 The effect of procedural programming

In section 3.3.2 we presented the *function* and the *sequentialization* patterns. We believe that these patterns are strongly related to the students' previous experience with procedural and sequential programming. In [18], Rist analyses the connection between the existence of plan schemas, and the design strategy one uses. When a plan schema that matches the problem at hand exists, a top-down design strategy is usually taken. When such a plan schema does not exist, a bottom-up strategy is taken, and as a result a new schema is built and stored in memory. Thus, a plan schema is developed in the context of a specific programming language and paradigm, and is therefore based on some underlying assumptions about the decomposition method, the execution model, and so forth.

The retrieval of schemas from memory is based upon the programmer's previous experience with similar problems. However, this model does not address the issue of using a plan schema that was developed in the context of one programming language when solving a problem in a different language. In the case where the schema is represented on an abstraction level that is higher than the relevant differences between the languages, the plan schema can be used as is. Otherwise, using the plan schema as is can lead to erroneous program behavior. In the case where the programmer is aware of the relevant differences, the model foresees that the programmer will use a more bottom-up approach to expand the schema. Our findings show that in some cases a different strategy is taken.

Since our students were experienced with procedural and sequential languages, it is reasonable to believe that they had a well-established execution model for procedural sequential languages, and that some of their plan schemas were developed in the context of such languages. As noted in Section 3.3.2, the LSC execution model is very different from the sequential and procedural execution model. When our students met familiar problems, they naturally tried to retrieve and apply existing plan schemas. In some cases, the assumptions underlying these schemas were not valid. When the students did not identify this gap, the result was an erroneous behavior. When the students did identify the gap, they tried to fill it. However, instead of expanding their schemas so they would fit the new paradigm and execution model, some students narrowed the power of the new language in a way that allowed them to use their plan schemas as is (this is reminiscent of the Greek legend of Procrustean, who either cut or stretched people's legs, so they would fit the size of an iron bed).

4.3 The effect of class:instance relationship

In section 3.3.3 we presented odd use patterns of *symbolic instances*. These patterns were mainly observed with students who had significant experience with C++/Java, which use the *class:instance* relationship as a main abstraction concept. We believe that the experience with this concept significantly influenced students' interpretation of symbolic instances. To base this argument, we show that students actually tended to interpret symbolic instances as an OOP pattern, and that the problems they had are related to the differences between the symbolic instances and classes.

Students' interpretation.

Students from group A tended to interpret symbolic instances as 'object-oriented'. For example, the following excerpt shows a spontaneous connection made by student #8:

> "I: And did the symbolic instances give you a convenient option to do things?
> S: Sure, it actually gave me a kind of object-oriented".

Also, students with OOP experience who were asked whether symbolic instances reminded them of any concept in another language, said that it reminded them of OOP. Though answers to direct questions might be biased, some students added spontaneous remarks that might indicate that not only did they couple the concepts, but they also perceive OOP as a significant and positive concept that influences their interpretation. Student #2:

> "S: [...] the good in the symbolic is that it elevates the object-oriented thing up, and that gives power..".

And student #5:

> "S: It [symbolic instances] is a little bit like class instantiation. You shouldn't think of it that way, but naturally the thoughts go to the things you are familiar with. [...] It's disrupting to think this way [as you are used to]..".

The EE student (experienced with procedural programming but not with OOP) said that symbolic instances did not remind her of anything.

Explaining the patterns.

The schema theory implies that experienced programmers will first try to apply available schemas when solving a programming problem. It is reasonable to believe that students with type A experience have design schemas that use the concept of class to achieve the goal of defining general behavior. When using LSC, they naturally applied these design schemas. One result was using the binding expression in a way that resembles the behavior of classes – the behavior is always applied to all the objects (what we call a meaningless binding expression). Another result was not using the concept of symbolic instances in a way that fully utilizes the extended expressivity power that this feature suggests. This was manifested in the meaningless use of the binding expression, and in the odd use of the existential vs. universal modality.

5. SUMMARY AND CONCLUSIONS

We have studied the influence of previous programming experience on the learning of new concepts introduced by the language of *live sequence charts* (LSC).

With respect to abstraction level, our findings indicate that moving from a low to a high-level programming language can lead programmers to feel that they lose power and control, and this feeling can cause them to develop a negative attitude that can affect their performance.

Our findings also demonstrate the long-lasting influence of experience in procedural and sequential languages. Learners not only interpret the new model through the prism of the previous models they are familiar with (this is the straightforward implication of constructivism), but they actively try to force the new model to behave like the model they are familiar with, so they can use previously acquired programming solutions. When the previous solutions do not behave in the new model as the programmer expected, this can lead to erroneous programs. The risk can be reduced if educators emphasize the execution model assumptions that underlie specific programming patterns. Another consequence is that when transforming from a sequential to a concurrent language, learners might continue to 'silently' build sequential programs 'under the hood' of a concurrent language, leading to poor use of powerful programming means. This has direct implications on the teaching of concurrent programming.

With respect to object-oriented programming, our findings suggest that experience with the *class:instance* relationship can hamper the learning of new abstraction concepts that aim for describing general behavior for groups of items, such as the *symbolic instances* of the LSC language.

More generally, the findings reported here provide empirical evidence and shed light on the ways in which previous programming experience influences the learning and use of new programming concepts.

6. ACKNOWLEDGMENTS

We would like to thank Moti Ben-Ari for commenting on a preliminary version of this paper.

7. REFERENCES

[1] B. Adelson and E. Soloway. The Role of Domain Experience in Software Design. *IEEE Trans. Software Eng.*, pages 1351–1360, 1985.

[2] R. C. Anderson. The Notion of Schemata and the Educational Enterprise: General Discussion of the Conference. In *Schooling and the Acquisition of Knowledge*. Lawrence Erlbaum, 1984.

[3] M. Armoni, J. Gal-Ezer, and O. Hazzan. Reductive thinking in computer science. *Computer Science Education*, 16(4):281–301, 2006.

[4] M. Ben-Ari. Constructivism in Computer Science Education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[5] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck. The Effect of Object-Oriented Programming Expertise in Several Dimensions of Comprehension Strategies. *CoRR*, abs/cs/0702002, 2007.

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

[7] F. Détienne. Design Strategies and Knowledge in Object-Oriented Programming: Effects of Experience. *Human-Computer Interaction*, 10(2-3):129–170, 1995.

[8] R. M. Felder and L. K. Silverman. LEARNING AND TEACHING STYLES IN ENGINEERING EDUCATION. *Engr. Education*, 78(7):674–681, 1988.

[9] J. Gal-Ezer and D. Harel. What (else) should CS educators know? *Commun. ACM*, 41:77–84, September 1998.

[10] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*, volume 3. Aldine, 1967.

[11] B. Haberman. High-School Students' Attitudes Regarding Procedural Abstraction. *Education and Information Technologies*, 9:131–145, 2004.

[12] 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.

[13] D. Harel and M. Gordon-Kiwkowitz. On Teaching Visual Formalisms. *IEEE Softw.*, 26:87–95, May 2009.

[14] 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.

[15] 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.

[16] O. Hazzan. Reducing Abstraction Level When Learning Abstract Algebra Concepts. *Educational Studies in Mathematics*, 40:71–90, 1999.

[17] M. Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, June 1995.

[18] R. S. Rist. Schema Creation in Programming. *Cognitive Science*, 13(3):389–414, 1989.

[19] R. S. Rist. Learning to Program: Schema Creation, Application, and Evaluation. In *Computer Science Education Research*. Routledge Falmer, 2004.

[20] A. H. Schoenfeld. *Mathematical problem solving*. Academic Press., Orlando, FL, USA, 1985.

[21] H. Sharp and J. Griffyth. The Effect of Previous Software Development Experience on Understanding the Object-Oriented Paradigm. *Journal of Computers in Mathematics and Science Teaching*, 18(3):245–265, 1999.

[22] J. Smith III, A. A. diSessa, and J. Roschelle. Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *The journal of the learning sciences*, 3(2):115–163, 1994.

[23] L. P. Steffe and J. Gale, editors. *Constructivism in Education*. Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 1995.

[24] M. Vujošević-Janičić and D. Tošić. THE ROLE OF PROGRAMMING PARADIGMS IN THE FIRST PROGRAMMING COURSES. *The Teaching of Mathematics*, XI(2):63–83, 2008.

[25] R. L. Wexelblat. The consequences of one's first programming language. *Softw., Pract. Exper.*, 11(7):733–740, 1981.