Scenario-Based Programming, Usability-Oriented Perception

GIORA ALEXANDRON, MICHAL ARMONI, MICHAL GORDON, and DAVID HAREL,
Weizmann Institute of Science

In this article, we discuss the possible connection between the programming language and the paradigm behind it, and programmers' tendency to adopt an external or internal perspective of the system they develop. Based on a qualitative analysis, we found that when working with the visual, interobject language of live sequence charts (LSC), programmers tend to adopt an external and usability-oriented view of the system, whereas when working with an intraobject language, they tend to adopt an internal and implementation-oriented viewpoint. This is explained by first discussing the possible effect of the programming paradigm on programmers' perception and then offering a more comprehensive explanation. The latter is based on a cognitive model of programming with LSC, which is an interpretation and a projection of the model suggested by Adelson and Soloway [1985] onto LSC and scenario-based programming, the new paradigm on which LSC is based. Our model suggests that LSC fosters a kind of programming that enables iterative refinement of the artifact with fewer entries into the solution domain. Thus, the programmer can make less context switching between the solution domain and the problem domain, and consequently spend more time in the latter. We believe that these findings are interesting mainly in two ways. First, they characterize an aspect of problem-solving behavior that to the best of our knowledge has not been studied before—the programmer's perspective. The perspective can potentially affect the outcome of the problem-solving process, such as by leading the programmer to focus on different parts of the problem. Second, relating the structure of the language to the change in perspective sheds light on one of the ways in which the programming language can affect the programmer's behavior.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer Science Education

General Terms: Languages, Human Factors

Additional Key Words and Phrases: Live sequence charts, psychology of programming

ACM Reference Format:
Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. 2014. Scenario-based programming, usability-oriented perception. ACM Trans. Comput. Educ. 14, 3, Article 21 (October 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2648814

A summarized version of this article appeared in *Psychology of Programming Interest Group (PPIG)*, 2011.
This research was partially supported by an advanced research grant to D. Harel from the European Research Council (ERC) under the European Community's FP7 Programme and by the Israel Science Foundation. The work of G. Alexandron was supported by a grant from the Azrieli Foundation.
Authors' address: G. Alexandron, M. Armoni, M. Gordon, and D. Harel, Weizmann Institute of Science, Rehovot 76100, Israel; email: giora.alexandron@weizmann.ac.il.

1. INTRODUCTION

At its core, programming[1] is about taking a problem defined in the *problem domain* and building a solution using the tools of the *solution domain*. This dichotomy poses several challenges. Among them is how to make sure that the usability requirements, which originated from the problem domain, are well addressed in the solution domain. Another challenge is how to reduce the cognitive effort involved in moving between the two domains. In the rest of this section, we review several aspects of these two challenges and briefly sketch the outlines of our argument, which suggests that the language of live sequence charts (LSC) [Damm and Harel 2001] fosters a kind of programming that addresses some of the issues related to the challenges just mentioned.

One of the factors that affect the usability of software artifacts is the development methodology. Over the years, many methodologies have been suggested, starting from the early *waterfall model* (e.g., see McConnell [1996]). One claim that was often raised against the early methodologies is that they neglect usability issues. To improve this, efforts were made in several directions. One of these is user-centered design methodologies, in which user needs are given high priority throughout the development process (the term *user-centered design* was used by Norman [2002]). Another direction is the development of more adaptive and flexible software engineering methods, such as *agile programming* [Beck et al. 1999].

Usability can also be affected by human factors that are not controlled by the methodology, such as the engineer's attitude. According to Jerome and Kazman [2005], it is common that R&D engineers make decisions regarding usability issues. However, some studies reported that software engineers tend to view usability issues as less important than the "real" parts of the system [Ferre and Medinilla 2007], and they consider the user interface as something that can be dealt with toward the end of the development process [Costabile 2000]. Programmers' attitude toward usability (and other) issues is probably affected by various factors, including individual characteristics and previous programming experience and education.

In the context of education, the IEEE/ACM computing curricula [2001, 2013] recommend allocating 8 core hours to HCI (and elective units are also suggested). Some researchers (e.g., Douglas et al. [2002]) have argued that this is not enough to achieve a satisfying coverage of this topic. Moreover, it seems that this recommendation is not always followed. For example, the institutes in which students who participated in this study earned their bachelor degree do not include HCI issues in their core hours. One domain in which usability issues arise is system-level development, especially when dealing with the kind of systems that include a significant component of user interaction (although we should emphasize that usability is a much broader issue than just interfaces). Such systems include household electronic goods, aerospace, automotive systems, and so forth. Thus, system-level development can be a natural and realistic context for introducing HCI. Another motivation for embedding high-level system design into the curriculum is that it emphasizes high-level programming (in terms of abstraction level). Several studies claim that dealing with high-level programming in early stages fosters the development of abstraction skills, and thus it is suggested to teach it in introductory courses (e.g., see the work of Scherz and Haberman [1995], Haberman and Kolikant [2001], and Warford [1999]).

The context of this research is system- and high-level programming, and the focus is on programmers' attitude during this activity. Specifically, the central question that we

[1]Throughout this article, we refer to programming in the broader sense, as an activity that involves both the high-level design and the implementation of an artifact. Thus, we usually refer to the one carrying out this activity as the *programmer*, but when the activity is mainly a design activity, we sometimes use the term *designer*.

study is the effect of the programming approach on programmers' tendency to adopt a more "user"- or more "programmer"-oriented perspective of the artifact at hand. A user perspective is associated with the problem domain, and a programmer perspective is associated with the solution domain. We believe that the perspective can have a significant effect on one's programming behavior, and from this, on the outcome of one's work—that is, on the final artifact. To illustrate the potentially significant effect of the perspective on one's performance, we examine its appearance in two theoretical frameworks that characterize problem-solving behavior and the factors affecting it. The first is the framework for problem solving of Schoenfeld [1985]. This framework was originally introduced in the context of mathematical problem solving but was later applied to other disciplines, including computer science (e.g., Ginat [1995]). It has four components: *resources*, *heuristics*, *control*, and *belief systems*. The perspective is relevant to the last three components.

Heuristics refers to strategies and techniques for making progress, and among other things includes testing and verification procedures. In the context of software development, the perspective can thus affect, for example, the methods that one uses to verify the artifact—whether on the basis of internal, engineering-oriented criteria or on the basis of external, usability-oriented criteria.

Control refers to global decisions regarding the selection and implementation of resources and strategies, such as planning and decision making. In the context of software development, one's bias toward a more user- or more programmer-oriented perspective can affect, for example, the allocation of resources to tasks.

Belief systems refers to the psychological aspects (not necessarily conscious) that affect performance, such as one's view about one's self or the environment. In the context of programming, holding a programmer-oriented perspective can lead to a viewpoint in which the designer knows what is right, thus requiring the users to change their behavior to accommodate the product. In contrast, a user-oriented perspective can lead to a viewpoint in which the product should accommodate the users.

The second framework is Wertsch's neo-Vygotskian interpretation of the zone of proximal development (ZPD) [Vygotsky and Cole 1978]. Wertsch [1984] defined the notion of *situation definition*. A situation definition is the way that the problem solvers capture and represent the task and the context in which they operate. Wertsch claimed that different participants (a child and an adult, in the context of the ZPD) might have a different situation definition, leading to actually not working on the same task. We believe that the perspective can influence the programmer's situation definition. Namely, a programmer who holds a usability-oriented perspective might define the task in a way that emphasizes the function that it fulfills and the ways in which it is used, whereas a programmer who holds an implementation-oriented perspective might define the task in a different way, emphasizing engineering issues. This can lead to each of them actually creating a different artifact as the consequence of the different situation definition. The ultimate goal of software engineering is supplying computerized solutions to real-world problems; therefore, holding a perspective that is more user oriented can lead programmers to create artifacts that are more suited to the needs of the user.[2]

We show that when working with the interobject, scenario-based language of LSC, programmers tend to adopt a perspective that is more user oriented than when working with an intraobject language, such as Statecharts. This is explained on two levels. First, the possible effect of the programming paradigm and the design methodology

[2]In this study, we concentrated on the perspective and did not evaluate its actual effect. We have some preliminary data indicating that indeed the different perspectives were associated with focusing on different aspects of the task, but it is not included in this article.

that accompanies it on the programmer's perspective is discussed. Second, a more comprehensive explanation, which resides in the cognitive domain, is suggested. It is based on a cognitive model of programming with LSC that we present, which results from the *behaviors* that Adelson and Soloway [1985] used to describe the role of domain experience in software design. Our model is an interpretation of their original model and a projection of it onto LSC. It describes how the combination of the three main concepts on which LSC is built supports a development process that allows programmers to spend less time in the solution domain and more time in the problem domain, so more attention is given to usability issues. By that, LSC might have the potential to address some of the issues related to the usability challenge mentioned at the beginning of this section.

However, the kind of programming that LSC fosters might also address the challenge of reducing the cognitive complexity. Tang et al. [2010] referred to moving between the problem and the solution domain as *context switching* and argued that extensive context switching reduces design effectiveness. Identifying the cognitive effort involved in context switching can be traced back to Dijkstra [1975], who claimed that "[T]his oscillation between 'the representation' and 'what it stands for' is an intrinsic part of the programmer's game, of which he had better be aware! (It could be also this oscillation, which makes programming so difficult for people untrained to switching between levels.)" (p. 1). This quote emphasizes that context switching is actually a special case of moving between levels of abstraction (the level of *the what* and the level of *the how*). This task is known to be difficult, especially for novices [Armoni 2013].

Considering that context switching arises when one needs to work both in the solution and in the problem domain, it is less prominent when dealing with low-level programming; however, it naturally comes up when dealing with high-level system design, as this requires referring both to the usability of the system and to its implementation. According to the model that we present in this article, LSC enables a kind of programming that requires less context switching. By that, we believe, it reduces the cognitive difficulty involved in system development (the language mainly aims to reactive systems development; see Section 2). Thus, it has the potential to be an appropriate vehicle for teaching high-level programming to advanced and novice students. The feasibility of this approach was examined in a pilot course on LSC that was given to high school students and followed the course described in the current study, as well as in several courses given to graduate computer science (CS) students. The research conducted on par with the teaching process suggested that high school and graduate students can reach a significant understanding of LSC [Alexandron et al. 2013a], and that when using it, students exhibit patterns of high-level thinking [Alexandron et al. 2013b]. Undoubtedly, implementing this approach in a larger scale requires further study.

To conclude, the main implication of this study is to non-novice students. Our study suggests that LSC can be used to raise the attention of such students to usability issues. Thus, it might be a good tool for dealing with HCI issues through system-level development on advanced stages of the curriculum. A less direct implication is to novice students. Our study suggests that LSC might be appropriate for teaching high-level programming on introductory levels in a way that also emphasizes HCI issues. We note that this implication emerges from the current study, but it was not empirically evaluated as part of it.

More generally, the results of this study shed light on an interesting human factor of programming—the programmer's perspective. To the best of our knowledge, issues related to this factor and to its relation to the programming language were not studied in the context of CS.

The rest of this article is organized as follows. In Section 2, we present LSC and its development environment, and discuss the differences between the approach behind LSC and the approach behind Statecharts [Harel 1987]. In Section 3, we describe the study and the findings. In Section 4, we present our model and discuss the findings in its light. We present our conclusions in Section 5.

2. LIVE SEQUENCE CHARTS

In this section, we introduce the language of LSC and its development environment, the Play-Engine. We review the three main concepts of the language and the Play-Engine: the underlying paradigm, which is also compared to the paradigm behind Statecharts, the play-in method, and the play-out method.

2.1. A Language for Reactive System Development

LSC and the Play-Engine present a novel approach to the specification, design, and implementation of reactive systems. Reactive systems are ones that continuously respond to stimuli of events from the external environment [Harel and Pnueli 1985]. Examples include control systems, household electronic goods, aerospace and automotive systems, communication networks, and so on. The complexity of reactive systems stems mainly from the intricate interactions between the system and its environment, as well as between the system components themselves, so the task of specifying the system behavior becomes complicated. To specify the behavior of reactive systems in a way that is formal and precise, temporal-logic (TL) languages, such as LTL [Pnueli 1977], were developed. However, such languages are less convenient as design tools and are mainly used for verification. LSC offers a specification language that is intended for design, without compromising rigor and formality. The language was originally introduced in Damm and Harel [2001] as an extension of message sequence charts (MSC) and was extended significantly in Harel and Marelly [2003]. LSC is supplemented with an operational semantics that enables one to execute/simulate the specification (see Section 2.4). Thus, LSC can actually function as a high-level programming language. A generalization of scenario-based programming, termed *behavioral programming*, was proposed in Harel et al. [2012]. Packages that support behavioral programming are available in Java, Erlang, C++, and Google's Blockly.

2.2. Scenario-Based Programming

2.2.1. The Paradigm and Interobject Specification. 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. For example, cash withdrawal is a basic functionality of an ATM. A scenario that describes the system behavior in cash withdrawal will describe the interactions between the person withdrawing money and the system, as well as between the internal parts of the system.

Since a scenario usually involves multiple objects—"one story for all relevant objects" (Harel and Marelly [2003], p. 4)—scenario-based programming is *interobject* by nature. Returning to the ATM, a scenario-based specification of an ATM will describe the ATM as a collection of such interobject scenarios. Scenarios in LSC are also *multimodal* in nature. That is, LSC events have behavioral properties that define modalities of behavior in three dimensions: execute versus observe, must versus may, and allow versus forbid. This allows the programmer to specify events that should be traced or executed, events that can or must occur, and events that are forbidden.

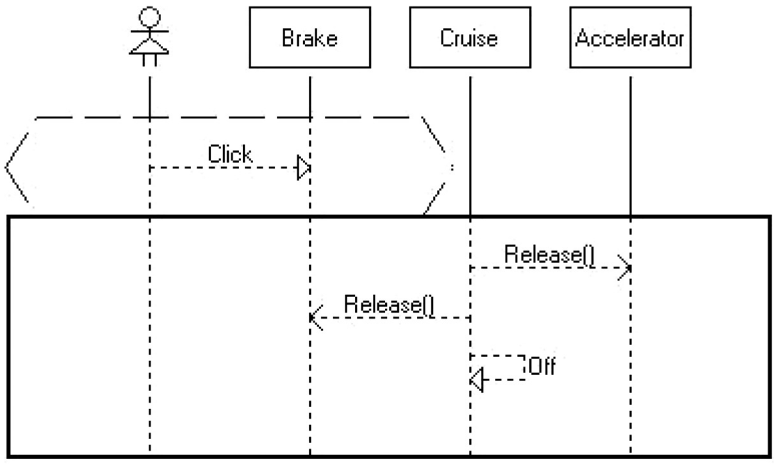

Fig. 1. LSC chart.

Syntactically, a scenario is implemented in an LSC (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 the event/s 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.

2.2.2. Statecharts, the Intraobject Approach, and Object-Oriented Programming. As opposed to the interobject approach, the more classical *intraobject* approach remains within the level of the object or the component. It describes the internal behavior of each object in the various states of the system—"all pieces of stories for one object" (Harel and Marelly [2003], p. 4)—and eventually describes the system as a collection of these objects.

Such an intraobject approach for specification is supported by many languages, and particulary by the visual language of Statecharts [Harel 1987], which is a part of the UML standard and is supported by various tools, such as IBM Rhapsody (http://www-01.ibm.com/software/awdtools/rhapsody/). Rhapsody can translate the Statecharts diagrams into C, C++, or Java. The translation of Statecharts into an object-oriented design is natural. Since object-oriented programming (OOP) is based on separating the system into objects and implementing each of the objects independently, it is intraobject by nature.

2.3. 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 and Marelly 2003], 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 approach allows the user[3] to apply his or her knowledge in a concrete manner, without the need to transform the knowledge into another representation (i.e., a programming language) to embed it into the system. Thus, users who are not familiar with LSC (or even with other programming languages) can program the behavior of an artifact relatively easily.

2.4. The Play-Out Method

LSC has an operational semantics that is realized by the *play-out* method (originally introduced in Harel and Marelly [2002]), which is also implemented in the Play-Engine. Play-out makes the specification directly execu/simulatable. When simulating the behavior, the user is responsible for carrying out the actions of the potential end user[4] and the external environment of the system. Play-out keeps track of the user/external actions and responds to them according to the specification. The play-out algorithm interacts with the system's GUI to reflect the state of the system on the fly. In addition, the Play-Engine presents a graphical representation of the state of all scenarios that are now active. For more details, see Harel and Marelly [2003].

3. THE STUDY

In this section, we present a qualitative study conducted as part of a larger research effort, in which we investigate educational issues involved in the learning of the language of LSC and its underlying paradigm, scenario-based programming (see Section 2).

The section is organized as follows. First, we present the research question. Second, we describe the methodology. Then, we present the analysis and the findings.

3.1. The Research Question

One of the issues that we study in the larger research effort is how working with LSC influences programmers' problem-solving behavior. The research question that we focus on in this article is whether working with LSC leads programmers to adopt a more user-oriented perspective. This question highlights one aspect of programming problem-solving behavior. As discussed in Section 1, this aspect can potentially have a significant effect on the programmer's problem-solving behavior.

3.2. Methodology

3.2.1. Research Approach. This study takes a qualitative approach to studying in-depth the difference between programmers' perception when using LSC and when using more conventional programming means. The strength of qualitative research lies in its ability to study complex phenomena in depth, and this can guide further, more focused research. It also has some known limitations, including its subjective nature and the fact that it (usually) relies on a small number of participants, and is executed in a specific context. These limit the generalizability of the results. Using postfactum interviews to analyze problem-solving behavior, as we do (see Section 3.2.4), has two known limitations. First, it refers to the thinking processes in retrospect. Second, the interview itself has an effect on the interviewees' behavior. Nevertheless, postinterview is considered a very valuable tool in qualitative research, among other things, because it yields deep insight into cognitive processes and attitudes. Thus, it is commonly used in CS education research, and in education research in general, in contexts

[3]We refer to the one using the Play-Engine as "user," although he or she is actually programming the system. This is to emphasize that this kind of programming does not necessarily requires one to be a "real" programmer.
[4]In the context of the Play-Engine, we use the term *end user* to denote the target user of the artifact.

that are similar to ours. In addition, we followed some steps to reduce undesired effects (e.g., by triangulating the results using two methods).

3.2.2. Definition of User/Programmer Perspective. Basically, a *user perspective* is a perspective of someone who is using the artifact. This perspective is external and usability oriented. Someone who is holding this perspective would probably be interested mainly in the usability of the artifact, would see the artifact as a means for achieving something, will focus on aspects of the problem that the artifact should solve, and would see the artifact as a "black box."

A *programmer perspective* is a perspective of someone who is building the artifact. It is an internal and implementation-oriented perspective. This perspective can be further characterized by referring to the artifact as a goal, focusing on aspects of the solution, seeing the artifact as a "white box," and constructing a mental model of the artifact.

We note that the categories defining each perspective are not necessarily orthogonal. For example, focusing on aspects of the solution can be manifested in dealing with implementation issues, and this requires seeing the artifact as a white box. We are fine with that, as our intention was not to supply the most concise definition, but to supply a description that captures various aspects of both perspectives.

This definition induced a set of criteria that constituted an operational definition used in the qualitative analysis. This set of criteria and the analysis based on it are described in Section 3.3.

3.2.3. 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 2010–11 at the Weizmann Institute of Science (see the course site: http://www.wisdom.weizmann.ac.il/~michalk/VisLang2011/). This course presented various aspects of reactive system development and concentrated on the two aforementioned approaches to the specification, design, and implementation of systems: the intraobject approach (through Statecharts) and the interobject approach (through LSC). About half of the course was devoted to Statecharts and the intraobject approach, and about half to LSC and the interobject approach.

Course assignments included two implementation projects, one of them to be carried out using Statecharts and the other to be carried out using LSC. This was the third time that this course was given (with some variations). A report on the first experience of teaching the course can be found in Harel and Gordon-Kiwkowitz [2009].

The student population was composed of graduate students studying toward an MSc or PhD in CS. Seven students taking the course participated in this study (out of nine students who took the course, two students could not be interviewed due to personal reasons unrelated to the course). Among these, four had a bachelor's degree in CS, two had a bachelor's degree in biology/bioinformatics, and one had a bachelor's degree in electrical engineering (EE). Some students had practical experience in the software industry, and some had only academic experience. The main programming experience of the students was with OOP, through C++ and Java, except for the EE graduate, who was mainly familiar with procedural programming in C. Students' age varied between (roughly) 25 and 35 years.

The rationale behind focusing on graduate students is that they are familiar with other programming approaches and are capable of designing and implementing high-level systems. Of course, this makes the results less generalizable to novice students. This issue is elaborated on in the discussion.

3.2.4. Data Collection Tools. The data collection tools included preinterview, class notes, postinterview, and a final project in LSC and in Statecharts. The postinterview was the main source of data for the analysis presented in this article, but it relied on the projects and on data collected at the preinterview.

(1) *Preinterview*: Each student was interviewed at the beginning of the course. The interview took about 45 minutes. In the context of this article, it was used to characterize students' previous programming experience (it also included other parts that are less relevant in this context).
(2) *Final projects in LSC and in Statecharts*: Students were directed to choose a system (of reasonable complexity) that they found appropriate and then implement it in both languages. Student projects included, among other things, modeling the blood's glucose level control system, modeling animal behavior, and modeling a variety of electronic devices. Students modeled the same parts of the system in both languages, except for one student who modeled a different part in each language. The part that this student implemented in Statecharts mainly included issues that are hidden from the user, and the part that the student modeled in LSC mainly dealt with HCI.
(3) *Postinterview*: Each student was interviewed at the end of the course. The interview took about 1 hour; it was semistructured and divided into two parts.

    In the first part, we asked the student about the LSC and the Statecharts projects. This included questions such as "Why did you choose this specific project?" "What differentiates your LSC project from your Statecharts project?" "Why did you make these specific design decisions?" and "How did you understand the main semantic idioms?" 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.

    In the second part of the interview, we asked the student to solve a programming task, using a think-aloud protocol. This part took between 20 and 30 minutes and was more open; during this process, we mainly observed the student's behavior while solving the task. The programming task was to design and implement (on paper) a computerized voting system for the primaries of a political party. Obviously, such a task cannot be completed in the time allocated for this part of the interview, but we were not interested in getting a complete solution, but in learning about the student's authentic approach. Thus, we did not give the student any specific guidelines on how to approach the problemand tried to keep the interference minimal. However, if we saw that the student missed some important issues or got stuck, we noted that, tried to figure out this issue with the student, and then guided the student to proceed. Once the student presented a high-level design and a system architecture that seemed sufficient for solving the problem, we stopped the student's work (some students did not reach this phase).

    The student's problem-solving behavior while working on the programming task and on the Statecharts project served as a reference point to which we compared the student's behavior while working in LSC. This is elaborated on in the next subsection. Overall, about 60% of the interview time was devoted to LSC and about 40% to the programming task and the Statecharts project.

3.2.5. Reference Point for the Comparison. To evaluate the change of perspective when solving a programming task with LSC, we needed a reference point—that is, an indication of the programmers' "normal" perspective when involved in a programming problem-solving task. The reference point that was taken was the students' perspective while working on the programming task given at the postcourse interview.

The rationale behind this choice is based on two assumptions. First, it is based on the assumption that the tasks we compared have similar characteristics. This gives us a sort of reliability. Second, it is based on the assumption that something like a normal perspective exists—that is, a perspective that is more or less consistent when the programmer is not constrained to use specific tools or focus on specific aspects of the problem. This gives us a sort of validity.

Regarding the characteristics of the programming task, the system that the students were asked to design had a significant component of user interaction, and the task focused on the design phase, which is done at a high level of abstraction. These factors characterize the kind of systems and the level of abstraction that LSC deals with.

The assumption that something like a normal perspective exists can be seen as an instance of a more general assumption that obviously underlies the framework of Schoenfeld [1985]. The framework assumes that, basically, problem-solving behavior depends on the set of relevant abilities that one holds. It does not mean that the environment has no effect on the behavior, but it does mean that there is some internal "state" that these abilities define. As explained in Section 1, the perspective is one of the relevant abilities that the framework captures. Of course, problem-solving abilities can evolve and change as one is involved in programming problem solving. Thus, it is reasonable that one's problem-solving abilities, including one's perspective, are strongly connected to one's previous programming experience, as shown in Alexandron et al. [2012].

To back up the comparison, we also used the perspective that the students demonstrated in the Statecharts projects. This was used in two ways. First, it was used as a triangulation point, which yielded another reference for the students' perspective when using a programming means different from LSC. Second, it gave us an empirical reinforcement for the assumption that programmers have something like a normal perspective. As noted, most of our students were mainly experienced with OOP, and we believe that this played a principal role in shaping their approach to programming. Thus, working in the context of OOP should reveal their normal perspective. As explained in Section 2.2.2, the intraobject approach that underlies Statecharts is closely related to the approach behind OOP languages. Thus, the perspective while working with Statecharts can be used to approximate the perspective when working with an OOP language. Combining these two arguments, we get that the perspective while working with Statecharts can be used to approximate the students' normal perspective. This conclusion is supported by the findings presented in Section 3.3, which show high similarity between the perspective when referring to the Statecharts projects and when working on the programming tasks.

To conclude, we believe that students' behavior in the programming task and in the Statecharts projects was a good approximation of their normal behavior when solving programming tasks in a general context. Thus, their perspective during the given tasks is a good approximation for their normal perspective and can serve as a reference point to which we can compare students' perspective when working with LSC.

3.3. Analysis and Findings

Next, we present our findings. The findings are divided into two sections. Section 3.3.1 describes findings obtained from content analysis, and Section 3.3.2 describes findings obtained from a more holistic qualitative analysis.

3.3.1. Content Analysis. The purpose of the content analysis was to give us a quantified measure as to the extent to which each student tended to adopt each of the two perspectives: the user perspective and programmer perspective. The content analysis followed the method for verbal analysis of Chi [1997].

The grain level for the analysis was "an argument." Usually, there was a one-to-one correspondence between an *argument* and a *turn*—that is, a specific answer to a question given by the interviewer. However, if a specific answer contained several different arguments, we coded each of the different arguments (this was quite rare). On the other hand, we did not code an answer to a question that was given as a follow-up to a previous answerif the answer merely continued the argument of the original answer. The criteria for classifying an utterance as a user or programmer perspective are given next. They were derived from the definition given in Section 3.2.2.

The criteria for coding an utterance as "user perspective" were:

(1) References to user needs as a parameter in decisions
(2) Descriptions that used terms taken from the problem domain (professional terms, tasks descriptions, external constraints)
(3) References to external features of the artifact (such as GUI)
(4) Use of external characteristics to describe the state of the artifact
(5) Use of use cases to verify the artifact
(6) Self-evidence of the student as holding a user perspective.

Following is an example of an utterance that was coded as a user perspective because of the obvious focus on the user side:

> "On the terminal you get a screen with a single option. The identification is by credit card or finger print. It can be a touch screen, or traditional voting by envelopes for old people who are not used to such interfaces. . . ."

The criteria for coding an utterance as a programmer perspective were:

(1) References to implementation issues as a parameter in decisions
(2) Descriptions that used terms taken from the solution space (architecture, data structures, etc.)
(3) Internal simulation of the artifact (as evidence for a mental model)
(4) References to aspects of the artifact that do not have an external expression, or "extra" investment in the implementation (e.g., in aesthetics of the program)
(5) Use of internal states to describe the state of the artifact
(6) Use of assertions and unit testing to verify the artifact
(7) Self-evidence of the student as holding a programmer perspective.

Next we present an example of an utterance that was coded as a programmer perspective because of the focus on the internal data structures that the program maintains:

> "You need a place for the temporary results. At the end it will be the final results. It's kind of a hash table, with the name of the person as the key, and the number of votes is the value. . . ."

The process was verified as follows. First, a sample of the data (one interview) was coded by the first and second authors. Second, the results were discussed until agreement was reached. Third, the rest of the data was coded by the first author.

The results of the content analysis are presented in Table I. We refer to the three activities on which we tested programmer perspective: the Statecharts project, the LSC project, and the programming task that was posed in the postinterview. For each activity, we show the number of utterances coded as user perspective (U) and the number of utterances coded as programmer perspective (P) for each student and for all students.

As can be seen in Table I, regarding the LSC activity, the total amount of utterances coded under each group is equivalent. This is more or less the situation with most of the students, except for student #4. For this student, the number of utterances coded as programmer perspective is much higher than the number of utterances coded as user perspective. This student had significant experience in low-level programming,

Table I. Comparing the Number of Utterances Classified as User or Programmer Perspective

	LSC		Statecharts		Programming task	
	U	P	U	P	U	P
Student #1	7	7	0	2	1	4
Student #2	6	6	0	2	0	5
Student #3	9	7	1	1	1	4
Student #4	2	7	1	0	1	1
Student #5	12	13	0	0	3	5
Student #6	8	5	1	2	0	2
Student #7	7	6	0	3	2	3
Total	**51**	**51**	**3**	**10**	**8**	**24**

such as in developing drivers. During the interview, this student revealed a negative attitude toward high-level programming. In Alexandron et al. [2012], the way in which this student and others accepted high-level abstractions was analyzed through the prism of the effect of previous programming experience. In the context of this article, it indicates that characteristics such as experience and individual preferences also affect programmers' perception. Regarding the Statecharts and the programming task activities, the results seem more biased toward the programmer perspective for most of the students, as well as in the total score.

However, we refer to these results only as an initial indication. First, the number of students was small. Although this is typical to qualitative research, it becomes more relevant when using quantified measures, as in this analysis. Second, looking only at small verbal fragments might fail in capturing perspectives expressed in a more holistic manner. In addition, the global goals of the interviews sometimes interfered with the more specific objective of looking into the issue of user/programmer perspective: when looking into the issue of user/programmer perspective, spontaneity is a major factor. However, one of the objectives of the interviews (in the context of the wider research and not in the context of this article) was also to collect data regarding students' understanding of the LSC semantics. Hence, some of the questions necessarily led the students to refer to the implementation details and "wear the programmer hat" when referring to their LSC project. This induced nonspontaneous utterances that were coded as programmer perspective.

3.3.2. Pure Qualitative Analysis. Based on the limitations of content analysis, we turned to a pure qualitative analysis. The analysis was conducted in two phases. In the first, a set of operational categories that stem from the user/programmer definition given in Section 3.2.2 was built. The categories were extracted from the data in a bottom-up manner, and each of them was defined as a dichotomy between the two perspectives. In the second phase, the data was analyzed thoroughly to find evidence corresponding to each of these categories. The analysis is presented next, by describing the categories and exemplifying them in order. This analysis sheds more light on students' perspective when working with LSC and with Statecharts, and during the solution of the programming task.

External- versus internal-oriented description of system processes. A user perspective is an external perspective of the system, whereas a programmer perspective is an internal perspective thereof. When describing their work with LSC, students revealed an external viewpoint that was reflected by the processes they chose to implement and the way they described them. This is exemplified by the following excerpt, in which student #3 describes his viewpoint when thinking about the system processes (here and following, I = interviewer, S*i* = student *i*):

> "S3: I considered the system and thought what processes can occur. The leopard is chasing them, they are running away . . . concrete scenarios that can occur, like I'm sitting there and watching it happen."

On the other hand, when working on the programming task, this same student described the system processes from an internal viewpoint and referred mainly to processes that occur "under the hood":

> "S3: [S]o the basis is the database, with some ESP script that will update the SQL table, and will take care of the synchronization."

In fact, we had to ask this student directly about the user interface because he did not make any spontaneous reference to it. (Later, he related this behavior to his previous experience with developing databases).

Focusing on aspects of the problem versus focusing on aspects of the solution. A user perspective focuses on the problem domain, whereas a programmer perspective focuses on the solution domain. When describing their work with LSC, students tended to focus on aspects of the problem domain. Among other things, this was evident in the jargon they used, which mainly included terms taken from the problem domain. For example, in the following excerpt, student #2 described a module of the biological system that her LSC project modeled:

> "S2: What happens is that each of the organs updates the glucose level, because of the hormone level, etc. Actually it takes glucose, breaks down glycogen to glucose, stores the glucose, and then when it takes glucose from the blood it reduces the glucose level. This happens with three organs."

However, when working on the programming task, the same student used jargon that was far more technical, mainly including terms taken from the solution domain:

> "S2: [T]here are on-line forms that you submit and get the input . . . instead of something that resides locally on your computer . . . something on-line where the server is located in one place and gets frequent updates . . . then you count the votes."

User-oriented design versus system-oriented design approach. With LSC, students' design approach focused on the usability aspects and on the user experience, whereas with Statecharts and in the programming task, their approach was much more system oriented. This was realized by the first aspects of the system that the students dealt with, which we interpreted as an indication of their perspective. For example, student #7 described the difference between the parts that she modeled in LSC and in Statecharts. Although with LSC the student started with the interface, with Statecharts the same student started with issues that are at the back end of the system:

> "I: And in LSC you did the same thing that you did in the Statecharts project?
>
> S7: No, in the Statecharts [project] I did transmission/reception, and in LSC I did HMI (Human-Machine Interface).
>
> I: And how is transmission/reception different from what you did in LSC?
>
> S7: In LSC I started with the user menus."

In the programming task, as with the Statecharts project, a system-oriented design approach was revealed by most students. For example, when giving the task to student #5, his first sentence was as follows:

> "S5: I suggest a network of terminals connected to a mainframe, or a network of computers with a server. All terminals are hidden from each other, and the communication is encrypted."

Focus on external aspects versus focus on system aspects when validating the artifact. When validating the LSC artifacts, it seemed that students tended to put more focus on external criteria. For example, student #1 was referring to how she verified her model by running a full simulation and comparing the results to the original biological model:

> "S1: Many organs release Glucose to the blood in a particular condition [in the biological model]. But we saw that only one is doing this [in the artifact], instead of all the organs. So we examined it carefully, and found that it was not running properly. . . ."

On the other hand, when validating the programming task, students tended to put more focus on system aspects, such the architecture, efficiency, and so forth. For example, student #5 described the kinds of problems that may have existed in his solution to the programming task. As can be seen, the focus of the student was on the system:

> "I: And what problems can occur?
>
> S5: There are many people, so I need the voting process to be short. . . . Now, if I don't maintain statistics [on the votes] then I only need a counter for each candidate, and if I do maintain statistics, then the database needs to be small. . . . You need power failure protection, in case of power outage."

Students reflecting on their perception. Finally, we also considered students' reflection on their perspective. During the interviews, we asked the students how they perceived their role with respect to the system they had developed. Most students were asked this question once, either in the context of LSC or in the context of the programming task. Interestingly, all students asked with respect to LSC reported a user-oriented perspective, whereas all those asked with respect to the programming task reported a programmer-oriented perspective. An exception was student #7, who exhibited a mixed approach. In the context of LSC, the student reported a user-oriented perspective, but in the context of the programming task, the student reported that she holds both perspectives simultaneously. Next, we give an example of one student reporting on his perception when working on the programming task, and the interesting answer of student #7 to this question in the context of LSC and in the context of the programming task. Student #3 was asked this question once, with respect to the programming task, and exhibited a programmer-oriented perspective. He also related this approach to his previous experience, strengthening our assumption that the programming task approximates students' general approach:

> "I: [A]nd when you consider that kind of question, do you think about it as someone who needs to implement the system, or as someone who needs to use it?
>
> S3: As someone who needs to implement it. It's a habit. I used to work on data bases, so I go straight to the solution and I don't think about the interface."

Student #7 was asked this question twice. When asked this question with respect to LSC, the student reported a user-oriented perspective:

"I: And when you worked on the radio interface, did you think as someone using it, or as the implementer?

S7: As the user."

In the programming task, this student was the only one who tried to use LSC. Interestingly, she was also the only student who gave a mixed answer when asked how she perceived herself with respect to the system in this context. Her answer revealed how she immediately translated the usability requirement to LSC scenarios, and how she captured the object-oriented implementation as a phase that is disconnected from the usability analysis. Her answer, although it is a little bit unclear (and maybe because of that fact), reflected on the connection between LSC and the tendency to adopt a user perspective:

"I: You looked at the question for a few minutes. What bothered you?

S7: About how to plan [the system]?

I: Yes.

S7: I thought of a scenario that inputs everything and then does some computation. But since you said 'many,' I thought that LSC is not efficient enough, at least for now.

I: So you imagined the people voting, how they would be using the system?

S7: Yes, the interface first. Someone comes, stands in front of the computer and votes. . . .

I: And then you thought about how to implement it?

S7: Yes.

I: And when you thought about how to implement it, did you think about the objects and their behavior?

S7: No, I thought of an LSC chart . . .

I: (Under the impression that the student is saying this because she believes that she was expected to use LSC) I don't mean that you should use a specific language. I'm interested in your considerations [in making these decisions]. When you thought about it, did you think as a user, or as the implementer?

S7: Both, at the same time. I don't know. As the one who comes to vote and as the implementer.

I: Do you mean that you switch between the perspectives?

S7: No, for me it's simultaneous . . ."

To summarize the findings, we found evidence that when working with LSC, programmers tend to adopt a more external and usability-oriented perspective than when working in a general context. These findings were triangulated from two directions: a content analysis of the transcripts of students' interviews and a pure qualitative analysis of students' interviews.

4. DISCUSSION

We now turn to explaining our results. First, we discuss the possible connection between the programming paradigm and the design methodology that accompanies it, as well as the programmer perspective. Second, we then present a cognitive model of programming with LSC, which supplies a more comprehensive framework for explaining the results. We then discuss implications to pedagogy.

4.1. The Effect of Paradigm

The intraobject approach that underlies the object-oriented paradigm focuses on the behavior of each object. Thus, a scenario must be broken down according to the boundaries of the participating objects. This yields an extra difficulty that stems directly from the specific solution domain and is not related to the problem domain. In contrast, scenario-based programming allows the programmer to capture an entire scenario in a single chart, so this extra effort is avoided.

This is demonstrated by the following excerpt, taken from the interview held with student #1 (the student referred to Statecharts through its development environment, IBM Rhapsody):

"I: What was the effect of the programming approach?

S1: In Rhapsody we needed to place the code in several locations and it was extremely unnatural. You need to think how to implement the scenario considering each of the objects. In LSC you just create one chart that captures it all."

After few sentences, the student spontaneously returned to this issue, this time referring to a specific piece of the system behavior:

"I: And are you looking at this LSC block as if it's equivalent to some states in the Statecharts?

S1: Yes.

I: And you are saying this block . . .

S1: Is divided across many Statecharts. It is an example of something that was very easy to implement in LSC.

I: Why?

S1: Because in the scenario you need to check the ratio once, and then you do the required operations, depending on the ratio.

I: And this check is equivalent to entering a state in Statecharts?

S1: But in Statecharts you need to check in many places, it's repeated four times, because the checks need to be independent."

This point is even reinforced if we consider that this student was very much experienced, and in fact worked on a daily basis, in Rhapsody, whereas LSC was quite new to her. Thus, we could expect that it would be easier for her to create an efficient design in Statecharts than in LSC. But this wasn't the case.

To summarize, it seems that with LSC, the programmer can spend less time and effort in the solution domain and in turn can invest more time and effort in the problem domain. This should result in a more user-oriented perspective.

4.2. The Effect of Design Methodology

A programming paradigm is usually accompanied with a corresponding design methodology that suggests how to carry out an effective development process. One of the common practices in object-oriented design is to start with writing use cases as a means for capturing the functional and behavioral requirements (e.g, see Jacobson [2004]). From these, the architecture of the system is derived. The internal behavior of each object is then implemented. This design methodology makes a clear separation between the problem domain and the solution domain. This can lead the programmer to give an extensive attention to the problem domain only at the beginning of the process, and afterward focus more on the implementation issues and neglect the usability issues. Thus, following the accepted object-oriented methodology can result in holding a more implementation-oriented perspective.

This is demonstrated in the following excerpt, taken from the interview held with student #7. This student was a very experienced object-oriented developer who was working in a leading IT company:

"I: So, you took some behavior, and modeled it? (referring to the work in LSC)

S7: Yes.

I: And do you work the same way with object-oriented [programming languages]?

S7: No, only when I do use-cases. . . .

I: So, you think of the use-cases and then transform them into objects?

S7: Yes, and then when they become objects I stop thinking about the scenarios. Maybe that's the problem . . ."

The preceding arguments suggest that scenario-based programming leads the programmer to think about the solution in terms of the external behavior, whereas the intraobject approach and the object-oriented methodology can lead the programmer to concentrate more on the internal behavior. They emphasize the role of the paradigm and the design methodology that accompanies it. However, we were looking for a more profound explanation that would be based on a cognitive mechanism that would suggest how the kind of programming that LSC encourages affects programmer perception.

4.3. A Cognitive Model of Programming with LSC

We now show how the main concepts behind LSC support a model of programming that describes how domain knowledge can be used for incremental construction of an artifact. Using this model, we can suggest an explanation for the way the characteristics of LSC promote user-oriented programming. This explanation is based on our interpretation of the model of Adelson and Soloway [1985] and a projection of it onto LSC. Next, we briefly review the original model and proceed to describe our take on it.

4.3.1. The Model of Adelson and Soloway. Adelson and Soloway [1985] observed expert and novice designers working on problems taken from familiar and unfamiliar domains. They described the main activities involved in the design process, which they called *Behaviors*, and studied the way the designers' experience affected the way those behaviors were carried out. The behaviors were described in a way that emphasizes their interconnection—that is, how one behavior facilitated the other. Together, the behaviors can be seen as creating a cognitive model of programming. In its essence, this model describes programming as an iterative refinement process, where at each stage the design-in-progress is simulated, and the gap between the results of the simulation and the expected results leads to the next refinement cycle. We now briefly describe these behaviors.

The first behavior is the *Formation of Mental Models*. According to Adelson and Soloway [1985], mental models "can be thought as the designer's internal design-in-progress" (p. 5), which supports internal simulation of the design. So, a working mental model is the basis for simulation.

The second behavior is the *Systematic Expansion of Mental Models*. The mental model starts at a very abstract level and becomes more accurate and complete as the design progresses.

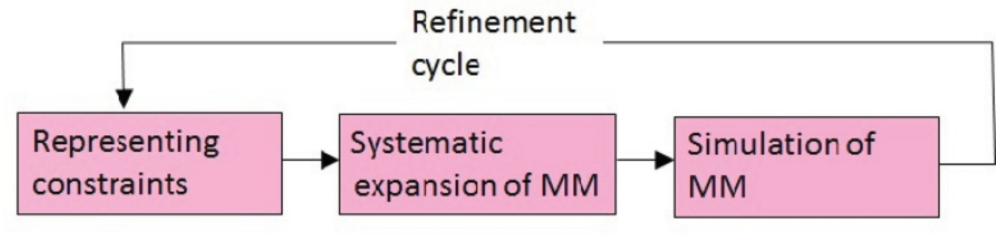

Fig. 2. The behaviors of Adelson and Soloway [1985] as forming a cognitive model of programming (MM, mental model).

The third behavior is the *Simulation of Mental Models*. The simulation allows the designer to assess the difference between the current state of the design-in-progress (held as a mental model) and the ultimate solution (the *goal state*). Then, proper steps can be taken to reduce the difference.

The fourth behavior is *Representing Constraints*. The constraints limit the mental representation of the problem, hence reducing the cognitive load by helping us concentrate on the important properties of the mental model.

We omit two other behaviors that we find less essential in this context: *Retrieving Labels for Plans* and *Note Making*.

The behaviors complete each other in the following manner: the constraints facilitate the creation of a working mental model, which is the key for simulation, and simulation allows one to realize what is missing in the design. This leads to another cycle in which the design is improved. This rationale is demonstrated in Figure 2. The first two behaviors that describe the formation and the expansion of the mental model are represented by the same box, of which its title is taken from the second behavior, as the first behavior is a sort of an initialization phase.

4.3.2. The Interpretation and Projection of the Original Model onto LSC. We claim that LSC allows the programmer to "imitate" this iterative refinement through the simulation process without the need to build a working mental model and simulate it. (This claim is taken to the extreme. We do think that the programmer holds an internal mental model of the artifact, but the issue is how detailed it is and how much it is essential in the process.) Hence, the programmer can enter the solution domain less often and spend more time in the problem domain. Our claim stems from a model of programming that connects the three main concepts behind LSC and explains how they complement each other. The model relies on the assumptions that we make on these concepts, but it also reinforces them. We now briefly describe the concepts, the assumptions that we make about these concepts, and the resulting model.

Scenario-based programming. Our first assumption is that scenario-based programming allows the programmer to think of the system they develop in the level of the external behavior. In Harel and Marelly [2003], it is argued that "when people think about reactive systems, their thoughts fall very naturally into the realm of scenarios of behavior" (p. 3; emphasis deleted). This observation is a main motivation behind the paradigm of scenario-based programming, on which LSC is built, and it is supported by empirical studies that refer to tasks as the way in which users think of the system they interact with. For example, Nardi [1993] emphasizes the importance of task-oriented programming languages for end-user programming. Gordon et al. [2012] show that even when given a language that is not specifically scenario oriented, novice programmers tend to decompose their system into scenario-like pieces, although they were not guided or taught to work this way. Ben-Ari and Yeshno [2006] state that end

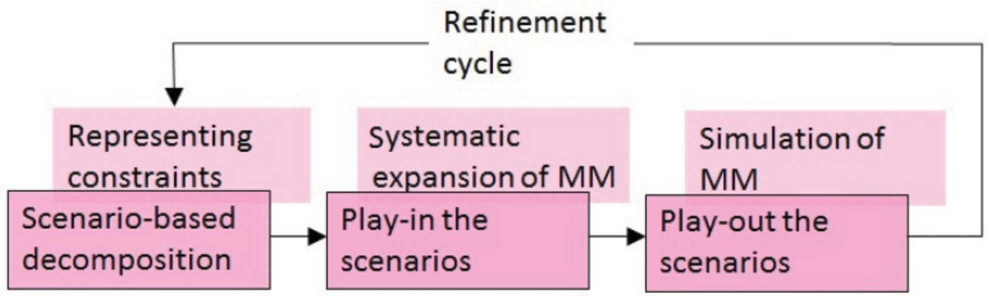

Fig. 3. A cognitive model of programming with LSC.

users' learning of artifacts like a word processor is mostly task oriented (although their claim is that such kind of learning falls short in the long term). Obviously, tasks are kind of scenarios that the system should fulfill. Finally, the postulate that users think through scenarios also underlies the accepted object-oriented design methodology, in which the design cycle starts by listing the use cases. Use cases define the external behavior of the system. Scenario-based programming is a programming paradigm that is based on decomposing the system into pieces of behavior—that is, use cases can be represented directly as code.

The play-in method. Play-in is a direct manipulation programming interface that enables to program the scenarios by "doing" them. We note that direct manipulation is a broad concept, but the basic idea is what Eisenberg [1995] calls *integrating mind-work and hand-work*. Norman [1991] argues that "the *naturalness* of a mapping is related to the *directness* of the mapping." In our context, this implies that a direct manipulation interface should be more natural. We interpret "natural" as something that is close to how the user thinks. Based on this, our second assumption is that play-in allows the user to add the scenarios to the artifact without the need to drop the user perspective and replace it with a programmer perspective.

The play-out method. The play-out method enables one to execute/simulate the artifact, compare the results to the expected behavior, and realize what behaviors should be added or refined. Our third assumption is that this can be done without the need to maintain an internal model of the artifact, because play-out executes the scenarios directly. This assumption stems directly from the first assumption that the scenarios capture the behavior in a way that is close to the way the user captures it.

The enhanced model. We now tie these three concepts together to form a cognitive model of programming with LCS, which stems from the model of Adelson and Soloway but replaces some of its components. The scenario-based paradigm allows us to decompose the system into scenarios, and the LSC language gives the syntactic idioms that enable the direct representation of these scenarios as code. In the model of Adelson and Soloway, this takes the place of *representing constraints* on the behavior of the artifact. Play-in allows one to add these constraints into the artifact, hence it enables expending the design-in-progress. In the original model, this corresponds to *systematic expansion of mental models*. Finally, play-out allows one to simulate the design-in-progress directly and to realize what behaviors should be added or refined. This takes the place of *simulation of mental models*.

The enhanced model is illustrated in Figure 3, together with the original model. Each component in the model is shown, with the corresponding component in the original model behind it in lightly colored shading. The arrows denote the interconnections

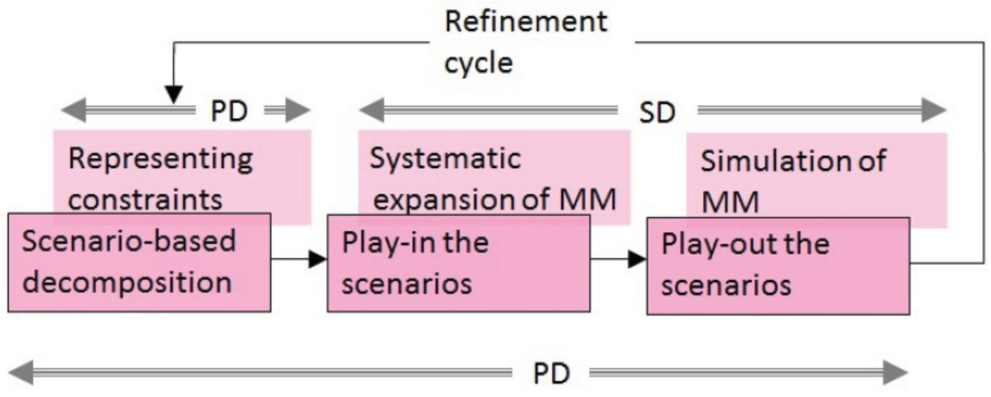

Fig. 4. Context switching. The arrows define the intervals that reside in the problem domain (PD) and in the solution domain (SD).

between the elements of the model. One iteration cycle goes through the phases from left to right, then a new refinement cycle begins.

To conclude, the three concepts together form a model of programming that enables a systematic expansion of an artifact by comparing the simulated behavior to the expected behavior, without the need to hold an internal mental model of the design-in-progress.

4.3.3. Interpreting the Results in Light of Adelson and Soloway's Model and Its Enhancement. Programming requires the programmer to work in both the problem and the solution domain. The model of Adelson and Soloway emphasizes that this is a cyclic, not linear, process: the programmer needs to consider user/external aspects when drawing constraints (which are derived from use cases, system requirements, etc.); then he or she needs to hold the programmer/internal perspective to construct the mental model and simulate it; then again, he or she needs to consider the user perspective when comparing the simulation to the expected (=external) behavior. The model of Adelson and Soloway puts the most weight on issues that are related to the solution domain—the way in which the mental model is built, expanded, and simulated.

The enhanced model suggests that with LSC, the overall rationale of iterative refinement through simulation remains the same, but that with LSC most weight is placed on behaviors that are related to the problem domain, and there is less context switching. This is illustrated in Figure 4. Although in the original model each iteration requires the programmer to switch between the problem and the solution domains, with the model that represents the work with LSC, much less context switching is required and more time is spent in the problem domain.

4.4. Summarizing the Discussion

The findings show that in LSC, students spent more time in the problem domain compared to the non-LSC tasks. An explanation that relates this to the underlying paradigm and to the methodology that accompanies it was suggested in Sections 4.1 and 4.2. However, the fact that the programmers spent more time in the solution domain in non-LSC tasks does not mean that they did more context switching on the specific task at hand. In fact, if the programmer remains in the solution domain, no context switching is required. However, remaining in the solution domain might result in neglecting usability issues, especially when dealing with systems in which the usability is a major issue (as opposed, say, to algorithms, in which the external behavior is usually defined in a very succinct manner). This emphasizes the role that the enhanced model takes in our overall argument: it explains how in LSC one can accomplish a programming task *that begins* in the problem domain, with less context switching, allowing the programmer to spend more time in the problem domain, which will hopefully lead to the creation of more usable artifacts.

Based on the discussion, we speculate that (1) LSC leads programmers to adopt a more user-oriented perspective, and (2) LSC suggests a programming means that enables realization of this perspective.

4.5. Implications to Pedagogy

The immediate implication of this study is to non-novice learners. This includes advanced undergraduate students, graduate students, and experienced programmers. All are also a target of CS education. The findings, and the model that explains them, suggest that programmers experienced in conventional programming languages tend to focus on the implementation and may neglect usability issues. According to the results, this tendency can be shifted (at least partially) by using certain programming means, such as LSC. We believe that this renders LSC and scenario-based programming in general good candidates for implementing some of the new units (core and elective) on HCI suggested in CC2013. In this context, we did not see a permanent shift and transfer of the usability-oriented perspective to other domains.[5] For that, a short experience (one course) is probably not enough for experienced learners who already have relatively established programming habits.

This brings us to the issue of novices, constituting a less immediate implication of the current study. One of the messages conveyed here is that we might want to consider HCI issues at early stages of the curriculum to assimilate the idea that usability is a central aspect of software engineering. Usability issues come up naturally when dealing with high-level programming and system-level development, so this kind of programming might be appropriate for teaching HCI. However, one of its difficulties relates to context switching, as it requires working and moving between the problem and the solution domain. Since LSC seems to enable a kind of programming that requires less context switching, it might reduce this difficulty. Thus, we believe that LSC and scenario-based programming might be appropriate for teaching HCI issues in the context of system-level programming to high school students. A proof of concept for this point was provided by a pilot course on LSC and scenario-based programming given in 2012 to 12th-grade high school students majoring in CS.

The study that was conducted on par with the teaching of the high school course indicates that teaching high-level programming with LSC might have another positive effect, which is encouraging abstract thinking [Alexandron et al. 2013b]. A main characteristic of system-level development is dealing with high-level abstraction. Implementing such systems using conventional, general-purpose programming languages, which demands defining the low-level behavior as well, requires moving between many levels of abstraction. This task poses considerable difficulty for novice programmers. The kind of programming that LSC encourages might reduce the cognitive difficulties caused by traversing between abstraction levels; therefore, it can help novice students deal with system-level programming.

In a broader context, the findings on the perception of the programmer shed light on an interesting cognitive aspect of software engineering. By that, they extend our understanding of the psychology of programming. This is relevant to CS education and also to SE domains such as tool development.

[5]This was not part of this study, and we did not collect data on it; however, as educators, we are always interested in whether students are able to generalize their knowledge.

5. SUMMARY

We have studied the question of whether working with the language LSC leads programmers to adopt a more user-oriented perspective. Our findings demonstrate that when working with LSC, programmers exhibit a perspective that is more user oriented than when working with conventional programming means. Relating these findings to the language itself, we presented a cognitive model of programming, suggesting that the combination of the main concepts behind LSC requires programmers to make less context switching between the problem and the solution domain and enables them to concentrate more on the former. To raise programmers' attention to usability issues, we suggest giving more weight to HCI and high-level design issues early in the curriculum. Based on the results of this study and a pilot course given to high school students, we believe that LSC and scenario-based programming can be a good vehicle for dealing with such issues with advanced students and might be also appropriate for novices.

Further research could explore two interesting questions: (1) What is the *actual* effect of holding a more usability- or a more programming-oriented perspective? In other words, what are the differences between artifacts built by students holding different perspectives? (2) How can we foster the transfer of the usability-oriented perspective from the context of LSC to other domains?

ACKNOWLEDGMENTS

The authors would like to thank Moti Ben-Ari, Rivka Taub, and Jason Cooper for useful discussions.

REFERENCES

Beth Adelson and Elliot Soloway. 1985. The role of domain experience in software design. *IEEE Transactions on Software Engineering* 11, 11, 1351–1360.

Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. 2012. The effect of previous programming experience on the learning of scenario-based programming. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling'12)*. ACM, New York, NY, 151–159.

Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. 2013a. On teaching programming with nondeterminism. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education (WiPSE'13).* 71–74.

Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. 2013b. Scenario-based programming: Reducing the cognitive load, fostering abstract thinking. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion'14)*. 311–320.

Michal Armoni. 2013. On teaching abstraction in CS to novices. *Journal of Computers in Mathematics and Science Teaching* 32, 3, 265–284. Available at http://www.editlib.org/p/41271.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. 1999. Manifesto for Agile Software Development. Retrieved August 4, 2014, from http://www.agilemanifesto.org/.

Mordechai Ben-Ari and Tzippora Yeshno. 2006. Conceptual models of software artifacts. *Interacting with Computers* 18, 6, 1336–1350.

Michelene T. H. Chi. 1997. Quantifying qualitative analyses of verbal data: A practical guide. *Journal of the Learning Sciences* 6, 3, 271–315.

Maria Francesca Costabile. 2000. Usability in the Software Life Cycle. In *Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing*, Hackensack, NJ, 179–192. (2000).

Werner Damm and David Harel. 2001. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* 19, 1, 45–80.

Edsger W. Dijkstra. 1975. About Robustness and the Like. Retrieved August 4, 2014, at https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD452.html.

Sarah Douglas, Marilyn Tremaine, Laura Leventhal, Craig E. Wills, and Bill Manaris. 2002. Incorporating human-computer interaction into the undergraduate computer science curriculum. *ACM SIGCSE Bulletin* 34, 1, 211–212. DOI:http://dx.doi.org/10.1145/563517.563419

Michael Eisenberg. 1995. Programmable applications: Interpreter meets interface. *ACM SIGCHI Bulletin* 27, 2, 68–93.

Xavier Ferre and Nelson Medinilla. 2007. How a human-centered approach impacts software development. In *Proceedings of the 12th International Conference on Human-Computer Interaction: Interaction Design and Usability (HCI'07)*. Springer-Verlag, Berlin, Heidelberg, 68–77.

David Ginat. 1995. Loop invariants and mathematical games. *ACM SIGCSE Bulletin* 27, 1, 263–267. DOI:http://dx.doi.org/10.1145/199691.199812

Michal Gordon, Assaf Marron, and Orni Meerbaum-Salant. 2012. Spaghetti for the main course?: Observations on the naturalness of scenario-based programming. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)*. ACM, New York, NY, 198–203.

Bruria Haberman and Yifat Ben-David Kolikant. 2001. Activating "black boxes" instead of opening "zippers"—a method of teaching novices basic CS concepts. *ACM SIGCSE Bulletin* 33, 3, 41–44.

David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8, 3, 231–274.

David Harel and Michal Gordon-Kiwkowitz. 2009. On teaching visual formalisms. *IEEE Software* 26, 3, 87–95.

David Harel and Rami Marelly. 2002. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and System Modeling* 2, 2, 82–107.

David Harel and Rami Marelly. 2003. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, Secaucus, NJ.

David Harel, Assaf Marron, and Gera Weiss. 2012. Behavioral programming. *Communications of the ACM* 55, 7, 90–100. DOI:http://dx.doi.org/10.1145/2209249.2209270

David Harel and Amir Pnueli. 1985. On the development of reactive systems. In *Logics and Models of Concurrent Systems*. Springer-Verlag, New York, NY, 477–498.

IEEE/ACM. 2001. Computing Curricula 2001: Computer Science—Final Report. Retrieved August 4, 2014, from http://www.acm.org/education/curric_vols/cc2001.pdf.

IEEE/ACM. 2013. Computer Science Curricula 2013 (Pre-release version). Retrieved August 4, 2014, from http://ai.stanford.edu/users/sahami/CS2013/final-draft/CS2013-Final-v0.9-prerelease.pdf.

Ivar Jacobson. 2004. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman, Redwood City, CA.

Bill Jerome and Rick Kazman. 2005. Surveying the solitudes: An investigation into the relationships between human computer interaction and software engineering in practice. In *Human-Centered Software Engineering—Integrating Usability in the Software Development Lifecycle*. Human-Computer Interaction Series, Vol. 8. Springer, Netherlands, 59–70.

Steve McConnell. 1996. *Rapid Development: Taming Wild Software Schedules.* Microsoft Press, Redmond, WA.

Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA.

Donald A. Norman. 1991. Cognitive artifacts. In *Designing Interaction*. Cambridge University Press, New York, NY, 17–38. Available at http://portal.acm.org/citation.cfm?id=120352.120354.

Donald A. Norman. 2002. *The Design of Everyday Things*. Basic Books.

Amir Pnueli. 1977. The temporal logic of programs. In *Proceedings of the Annual IEEE Symposium on the Foundations of Computer Science*. 46–57. DOI:http://dx.doi.org/10.1109/SFCS.1977.32

Zahava Scherz and Bruria Haberman. 1995. Logic programming based curriculum for high school students: The use of abstract data types. *ACM SIGCSE Bulletin* 27, 1, 331–335.

Alan H. Schoenfeld. 1985. *Mathematical Problem Solving*. Academic Press, Orlando, FL.

Antony Tang, Aldeida Aleti, Janet Burge, and Hans van Vliet. 2010. What makes software design effective? *Design Studies* 31, 6, 614–640.

Lev S. Vygotsky and Michael Cole. 1978. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press.

J. Stanley Warford. 1999. BlackBox: A new object-oriented framework for CS1/CS2. *ACM SIGCSE Bulletin* 31, 1, 271–275. DOI:http://dx.doi.org/10.1145/384266.299785

James V. Wertsch. 1984. The zone of proximal development: Some conceptual issues. *New Directions for Child and Adolescent Development* 1984, 23, 7–18. DOI:http://dx.doi.org/10.1002/cd.23219842303

Received May 2013; revised May 2014; accepted May 2014