Programming with the User in Mind *

Giora Alexandron[1] Michal Armoni[1] David Harel[1]

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

Keywords: POP-II.B.scenario-based design, POP-III.C.visual languages, POP-III.D. specification languages, POP-V.A.problem solving

Abstract. In this paper we present preliminary findings regarding 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. According to the findings, when working with the visual, inter-object language of *live sequence charts (LSC)*, programmers tend to adopt an external and usability-oriented view of the system, while when working with a language that is intra-object in nature, they tend to adopt an internal and implementation-oriented view of the system. To explain the findings, we present a cognitive model of programming that is based on that of Adelson and Soloway [1]. Our model suggests that the new paradigm of *scenario-based programming*, upon which LSC is based, combined with concrete interface programming and the ability to directly simulate the scenarios, allows the programmer to build systems while concentrating more on the user side. This work has two main implications. First, we believe that our findings on the programmers' viewpoint are interesting in themselves. Second, it sheds light on how the LSC approach supports programming that requires less work in the solution domain. This can be applicable in areas such as novice and end-user programming.

1 Introduction

Programming [1] is the main activity through which computerized solutions to real-world problems are built. To construct a usable artifact, the programmer [2] first needs to determine what the system should do, who the user of this system is, what the interaction between the user and the system should look like, and so on. These parameters belong to what is defined in the software engineering (SWE) literature as the *problem domain*. Once these "external" characteristics are defined, the programmer can go on to design and implement the internal structure of the artifact. This activity occurs within what the SWE literature defines as the *solution domain*.

Though the transfer from the problem domain to the solution domain is usually presented as a linear process, in practice it is more cyclic, as demonstrated by the model of Adelson and Soloway [1]. The design and the implementation phases bring up issues that were not considered earlier, point to ambiguous requirements, and so on. This sends the programmer back to the problem domain, and so on. In [17], Tang et. al. refer to moving between the problem and the solution domains as *context switching*. They argue that extensive context switching reduces software design effectiveness. Also, the comparison Adelson and Soloway make between novice and experts in [1] might indicate that novices are less capable of moving back and forth between the problem domain and the solution domain. Hence, context switching has a negative effect on performance in design tasks, but in particular on the performance of less experienced programmers. The conclusion is that all programmers, especially the less experienced ones, could benefit from programming means that require less context switching.

In this paper we present our preliminary findings regarding the possible connection between the programming language one uses and the tendency to adopt a more 'user'- or a more

* 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).

[1] Throughout this paper, we refer to programming in the broader sense, as an activity that involves both the design and the implementation of a computerized artifact.

[2] We refer to programmer as the one who carries out the activity of programming, as defined above.

'programmer'-oriented perspective of the artifact one builds. Our findings show that when working with the language of *live sequence charts* (LSC) [4], programmers tend to adopt a more user oriented perspective of the system they develop. After presenting the findings, we suggest a cognitive model of programming that stems from the *behaviors* that Adelson and Soloway [1] use to describe the role of domain experience in software design. Our model relates the findings to the three main concepts of LSC, and suggests that together these concepts form a programming means that enables the programmer to spend more time in the problem domain, and also reduces the number of context switchings.

The rest of this paper is organized as follows. In the next section we present LSC and its development environment, and compare the approach behind LSC with the approach behind Statecharts [6]. In Section 3 we describe the study, present our model and discuss the findings in its light. We present our conclusions in Section 4.

2 Live sequence charts

In this section we introduce the language of *live-sequence charts* (LSC) and its development environment, the *Play-Engine* (PE). We review the three main concepts of the language and the PE: 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

Live sequence charts and the Play-Engine present a novel approach to the specification, design and implementation of reactive systems [11]. The language was originally introduced in [4] and was extended significantly in [7]. Reactive systems are ones that continuously respond to stimuli of events from the external environment [11]. 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 very complicated. In order to specify the behavior of reactive systems in a way that is formal and precise, temporal-logic (TL) languages, such as LTL [15], 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. Like temporal-logic languages, LSC deals with specifying system behavior over a given system model, and does not deal with designing the system itself. LSC is supplemented with an operational semantics that enables one to execute/simulate the specification (see section 2.4). Thus, LSC actually functions as a high-level programming language.

2.2 Scenario-based programming

The paradigm and inter-object 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, 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. A scenario that describes the system behavior in cash withdrawal, will describe the interactions between the person withdrawing money and the system, and between the internal parts of the system.

Since a scenario usually involves multiple objects — "one story for all relevant objects" ([10], p. 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, and among other things can specify events that can or must occur, as wells ones that are forbidden

Syntactically, a scenario is implemented in a live sequence chart. Roughly speaking, the chart consists of vertical lines ('lifelines') that represent objects, and horizontal lines that represent interactions between the objects, where the flow of time is top down (see Figure 1), messages between them, and conditions. The visual representation emphasizes the general structure of the scenario, the flow of control and the interactions between the objects.

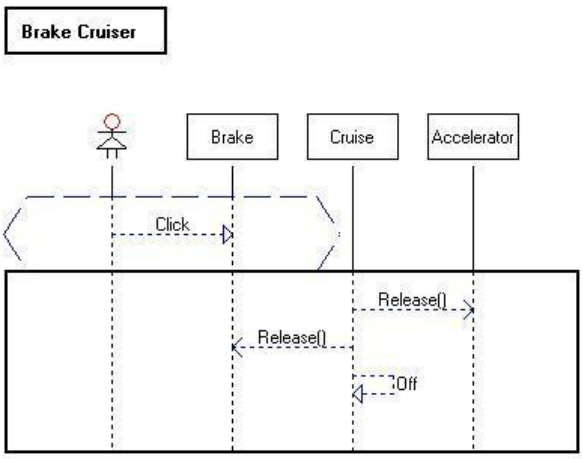

Figure 1. A simple LSC describing a scenario of stopping the cruise control in a car.

Statecharts, the intra-object approach and OOP. As opposed to this inter-object approach, the more classical *intra-object* 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" ([10], p. 4) — and eventually describes the system as a collection of these objects.

Such an intra-object approach for specification is supported by many languages, and in particulary by the visual language of Statecharts [6], 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, e.g., C, C++ or Java. The translation of Statecharts into OO is natural. Since OOP is based upon separating the system into objects and implementing each of the objects independently, it is intra-object 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 [10]. 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/her knowledge in a concrete manner, without the need to transform the knowledge into another representation (i.e., a programming language) in order to embed it in 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.

[3] We refer to the one using the PE as 'user', though he/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.

2.4 The play-out method

LSC has an operational semantics that is implemented by the play-out method (originally introduced in [9]), and is also implemented in the Play-Engine. Play-out makes the specification directly executable/simulatable. When simulating the behavior, the user is responsible for carrying out the actions of the potential end-user[4] and the 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 GUI to reflect the state of the system on the fly. Also, the PE presents a graphical representation of the state of all the scenarios that are now active. For more details see [10].

3 The Study

In this section we present an empirical study conducted as part of a larger research effort, in which we investigate educational issues involved in the learning of the language of *live sequence charts* (LSC), and its accompanying paradigm, scenario based programming (see section 2). The findings reported upon here are taken from the pilot phase of this larger effort. The section is organized as follows. First, we present the research question. We then describe the research setting, which includes the course that was at the basis of the work, the population and the data collection tools. We describe the analyses that we carried out, and their results, and, finally, we present our enhancement of the model of Adelson and Soloway [1] and discuss the findings in its light.

3.1 The research question

One of the issues that we study in the larger research effort is how working with LSC affects the way programmers solve programming problems.

The research question that we investigate in this paper is whether working with LSC leads programmers to adopt a more user-oriented perspective. This question highlights one aspect of programming problem solving behavior. In Schoenfeld's framework for problem solving [16], this aspect falls into the categories of *belief systems*, i.e., the psychological aspects that affect one's performance: *control*, which refers to the meta-cognitive tasks involved in problem solving, and *heuristics*, which refers to strategies and techniques for making progress.

3.2 Research setting

The setting of the study was based on the course "Executable Visual Languages for System Development", given by the third-listed author in the Fall term of 2010-2011 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 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 two implementation projects, one of them to be carried out using Statecharts, and the other 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 both languages. Students projects included, among other things, modeling the blood's glucose level control system, modeling animals behavior, and modeling a variety of electronic devices. Students were allowed to choose between modeling exactly the same parts of the system in both languages, or modeling one part of it in Statecharts and the other in LSC. With some variations,

[4] In the context of the PE we use the term end-user to denote the target user of the artifact.

this course is given for the third time. A report on the first experience of teaching the course can be found in [8].

The student population was composed of graduate students studying towards an M.Sc or Ph.D in computer science. Seven students taking the course participated in our study. Among these, four were CS graduates, two had degrees in Biology/Bioinformatics, and one in 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.

Data collection tools included:
i) Pre-interviews: These were mainly used to characterize the student's previous programming experience. During the interviews, we also gave each student a programming task, which is less relevant in the context of the present paper.
ii) Student presentations: Each student presented his/her project to the course team.
iii) Student projects: After the presentations, we analyzed the LSC projects in order to be familiar with them before the post-interviews, and to identify interesting patterns.
iv) Post interviews: Each interview was divided into two parts. In the first part, we asked each student about the LSC project, and in the second we asked the student to solve a programming task, using a think-aloud protocol.

The post interviews were semi-structured. The first part included the following questions to the student: Why did you choose this specific project? What differentiates your LSC project from your Statecharts project? Why did you take this specific design decisions? 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.

The second part was more open, during which we mainly followed-up on the student's behavior while solving a programming task with minimal interference. However, if we saw that the student missed some important issues, we noted that, and tried to figure out with the student why this issue was neglected. Also, if the student got stuck, we noted that as well and guided the student to progress. Once the student presented a design that seemed sufficient for solving the problem, we stopped the student's work. (Some of the students were not able to create a satisfactory solution.) The interviews were held in Hebrew, and were conducted by the first-listed author. They were fully transcribed, to facilitate future analysis.

The primary source of data for the analysis presented in this paper were these post-course interviews. Since the post-interviews were mainly held around the projects, being familiar with the projects before the interviews was an obvious preparation step. The presentations helped with that, providing us with a brief summary of each project, so it was easier to 'dive into' the projects for further analysis. The pre-course interview helped us become familiar with the students and their background. This assisted us both on the interpersonal level, and in interpreting students' answers and behavior.

3.3 Analysis

Our conjecture was that working with LSC leads programmers to adopt a user-oriented perspective. The goal of the analysis was to check if the empirical data significantly supported this conjecture.

The analysis is presented as follows. We define the reference groups, and then give an accurate operational definition for a user/programmer perspective. Following this, we use a content analysis protocol on the transcripts of the interviews, in order to obtain quantitative measures, and then analyze the interviews using a purely qualitative method.

Reference group. The perspective of the students while working on the programming task given at the post-course interview was used as a reference point; this was an indication of stu-

dent's 'normal' perspective when working on programming tasks. In the task, the students were not directed to choose any specific methodology or tool, so they were free to choose how to approach the problem. Thus, we believe that the past programming experience was the main factor that affected the student's problem solving behavior. The task itself and the way it was executed created a built-in bias towards user perspective:
i) The given task included a significant component of user interaction. This should have encouraged the students to refer to the usability of the system.
ii) We stopped students who presented a sufficient design, and did not ask them to actually implement the system. Obviously, the design phase requires much more usability analysis than the implementation phase.
This bias should strengthen the significance of the difference between the two groups, if found.

Statecharts as a special case. The Statecharts projects were used as a triangulation means for the student's regular perspective. This is a special case, because Statecharts is a specific programming language and we have no data on the influence of working with this language on a programmer's perspective. However, in light of the intra-object nature of Statecharts, and its correspondence with OOP, it is reasonable to believe that working with Statecharts can be a good approximation to working with an OO programming language. Since most students' main programming experience is with OOP, we believe that there should be a good correlation between the Statecharts results and the post-course interview programming task results.

Operational definition of user and programmer perspective. To operationalize each perspective, we defined it by means of a few categories. Recall that we refer to programming in the broader sense, as an activity that includes both the design and implementation of the solution. 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*.

A user perspective is an external and usability-oriented one. It is characterized by referring to the usability of the designed system (the artifact), referring to the artifact as a means, focusing on aspects of the problem that the artifact should solve, seeing the artifact as a 'black-box', and verifying it using external criteria.

A programmer perspective is an internal and implementation-oriented one. It is characterized by referring to the artifact as a goal, focusing on aspects of the solution, seeing the artifact as a 'white-box', constructing a mental model thereof, and verifying it using internal criteria.

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 Chi's method for verbal analysis [3].

We coded as 'user perspective' utterances that included references to user needs as a parameter in decisions; descriptions that used terms taken from the problem domain (professional terms, tasks descriptions, external constraints); references to external features of the artifact (such as GUI); the use of external characteristics to describe the state of the artifact; the use of use-cases to verify the artifact; self evidence of the student as holding a user-perspective.

We coded as 'programmer perspective' utterances that included references to implementation issues as a parameter in decisions; descriptions that used terms taken from the solution space (architecture, data structures, , etc.); internal simulation of the artifact (as evidence for a mental model); references to aspects of the artifact that do not have an external expression; 'extra'-investment in the implementation (for example, in 'aesthetics' of the program); the use of internal states to describe the state of the artifact; the use of assertions and unit testing to verify the artifact; self evidence of the student as holding a programmer-perspective.

The grain level for the analysis was 'an argument'. Usually, there was a one-to-one correspondence between 'an argument' and 'a turn'; i.e., 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 previous answer, if the answer merely continued the argument of the original answer.

The results of the content analysis are presented in Table 1, where we refer to the three activities on which we tested student's perspective: the Statecharts project, the LSC project, and the programming task that was posed in the post interview. For each activity, we show for each student the number of utterances coded as 'user perspective', and the number of utterances coded as '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

Table 1. Comparing the number of utterances classified as User or Programmer perspective.

As can be seen in Table 1, regarding the LSC activity, the amount of utterances coded under each group is almost equivalent for most students, with the exception of student 4 and student 6. For student 4, the number of utterances coded as 'programmer perspective' is much higher than the number of utterances coded as 'user perspective'. For student 6, the number of utterances coded as 'user perspective' seems significantly higher than the number of utterances coded as 'programmer perspective'. Regarding the Statecharts activity, and even more for the programming task activity, the results seem more biased toward the 'programmer perspective'.

However, we can refer to these results only as an initial indication. First, the number of students was small. Second, the verbal analysis method has inherent limitations: looking only at 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, spontaneousness 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 paper) 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'. This induced non-spontaneous utterances that were coded as 'programmer perspective'.

Pure qualitative analysis. Based on the limitations of content analysis, we turn to a pure-qualitative analysis, which is focused on three categories: The first is the students' self reflection on their perspective. During the interviews, there were questions that enabled the students to express their subjective opinion regrading the perspective that they held with respect to some of the activities. The two remaining categories are taken from the set of categories included in the operational definition of user/programmer perspective. (For lack of space, we exemplify only two categories from the operational definition). By mapping student utterances according to these three categories, we shed more light on their perspective when working with Statecharts, with LSC, and during the solution of the programming task. This mapping is demonstrated by the excerpts of a few students.

Students reflecting on their perception. During the interviews, we asked the students how they perceived their role with respect to the system they had developed. Each student was asked this question once, in the context of either LSC or in the context of the programming task. The results were remarkably consistent: all the students who were asked this with respect to LSC reported a 'user-oriented' perspective, while all the students who were asked this with respect to the programming task reported a 'programmer-oriented' perspective. This is demonstrated by the answers of two of the students (I = interviewer, S = student):

Student 3 was asked this question with respect to the programming task:
I: "[...] and do you think of yourself as the one who needs to implement this system, or as the one who needs to use it?"
S3: "As the one who needs to implement it. It's a habit. I used to develop data bases, so I go straight to the solution and I don't think of the interface."

Student 7 was asked this question with respect to LSC:
I: "And when you worked on the interface of the radio, did you think of it as the one who uses the radio, or as the one who implements it?"
S7: "As the user."

External vs. internal oriented description of system processes. When describing their work with LSC, the students revealed an external viewpoint of the system. Their answers also demonstrated how the external viewpoint is reflected through scenarios. This is exemplified by the following excerpt, taken from the interview with student 3.
S3: "I've observed the system and thought what processes can occur. The leopard is chasing after them, they are running away... perceptible scenarios that can happen, like I'm sitting there and watching it."
On the other hand, when working on the programming task, this student mainly referred to the internal parts of the system (in fact, we had to ask this student directly about the user interface):
S3: "[...] so the basis is the data base, with some ESP script that will update the SQL table, and will take care of the synchronization."

Focusing on the aspects of the problem vs focusing on aspects of the solution. When describing their work with LSC, the students tended to focus on aspects of the problem domain. In the following excerpt, student 2 described a certain property 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 hormones level, etc. Actually it takes glucose, disassembles glycogen to glucose, accumulates 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 student's approach is much more solution oriented:
S2: "[...] there are on-line forms that you insert and gets the input... instead of something that sits locally on your computer... something on-line that the server sits in one place and gets frequent updates... then you count the votes. Is that what you meant?"
I: "Maybe, I don't know how these things work."
S2: "Many votes from different..." [the student is not sure how to proceed with the solution.]
I: "And when you think of this question, do you think as the one who needs to use it or..."
S2: [interrupting the interviewer] "No, as the one who needs to implement it."

To summarize the findings, there is evidence that working with LSC leads programmers to adopt a more external and usability-oriented perspective. These findings were triangulated from two directions: a content analysis of the transcripts of students' interviews, and a pure qualitative analysis of students' interviews.

3.4 Discussion

We now turn to explaining our results in light of the paradigm upon which LSC is based. The two following examples illustrate how the paradigm and the methodology that accompanies it can affect the programmer's perspective.

The effect of paradigm. Because the intra-object approach focuses on the behavior of each object, 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: "[...] there were places in Rhapsody were we had to divide the code between several places and it was extremely unnatural, you need to think how to make the scenario while referring to each object. In LSC you just make one chart that captures it all."

Thus, with LSC the programmer can spend less time and effort in the solution domain, so he/she can invest more time and effort in the problem domain. This should result in a more user-oriented perspective.

The effect of methodology. A programming paradigm is usually accompanied with a corresponding methodology that suggests how to carry out an effective development process. One of the most used OO methodologies starts with writing use-cases as a means for capturing the functional and behavioral requirements (see for example [12]). From these, the architecture of the system is derived. The internal behavior of each object is then implemented. This methodology makes a clear separation between the problem domain and the solution domain, and the result can be that the programmer will eventually focus on implementation issues and will neglect the usability issues.

This is demonstrated in the following excerpt, taken from the interview held with student 7. This student is a very experienced OO developer, who is working in a leading IT company.
I: "So, you took a behavior, and modeled it?" (referring to the work in LSC)
S7: "Yes."
I: "And do you work in the same way when you work with OO?"
S7: "No, only when I do use-cases."
I: "So, you do use-cases and then transfer them into the objects?"
S7: "Yes, and then when it becomes objects I stop thinking on the scenarios. Maybe that's the problem..."

Thus, following the accepted OO methodology can lead the programmer to adopt a more implementation-oriented perspective.

These two examples suggest that scenario-based programming leads the programmer to think about the solution in terms of the external behavior, while the intra-object approach and the OOD methodology can lead the programmer to concentrate more on the internal behavior. This explanation emphasizes the role of the paradigm (and the methodology), but programming is much more than just problem decomposition.

To provide a more complete explanation, we 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. The model we suggest stems from that of Adelson and Soloway [1]. We first briefly review that model and proceed to describe our enhancement thereof. We then provide a possible explanation for our findings based on the enhanced model.

The model of Adelson and Soloway Adelson and Soloway [1] 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; i.e., how one behavior facilitated the other. Together, the behaviors can be seen as creating a 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 simulation results 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*. Mental models "can be thought as the designer's internal design-in-progress" ([1], 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.

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 forth 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; simulation allows one to realize what is missing in the design. This leads to another cycle in which the design is improved.

Our enhancement of Adelson and Soloway's model We claim that LSC allows the programmer to 'imitate' this iterative refinement through simulation process without the need to build a working mental model and simulate it[5]. 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 (SBP). Our first assumption is that SBP allows the programmer to think of the system they develop in the level of the external behavior. In ([10], p. 3), it is argued that "when people think about reactive systems, their thoughts fall very naturally into the realm of **scenarios of behavior**" (emphasized in the source). This is also supported by studies that refer to tasks as the way users tend to think of the system they interact with. For example, Nardi [13] emphasizes the importance of task-oriented programming languages for end-user programming. Ben-Ari and Yeshno state in [2] that end-users' learning of artifacts like a word processor is mostly task-oriented (though 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. The postulate that users think through scenarios also underlies the accepted OOD methodology, in which the design cycle starts by listing the use cases. Use cases define the external behavior of the system. SBP is a programming paradigm that is based on decomposing the system into pieces of behavior; i.e., use cases can be represented directly as code.

[5] This statement takes this claim to the extreme. We do think that the programmer holds an internal mental model of the artifact, but the question is how much this model is essential in the process.

The play-in method 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 ([5], p. 5) calls "Integrating Mind-work and Hand-work". Norman [14] 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 enables us 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 model that 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*.

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

Interpreting the results in the 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 a linear, process: The programmer needs to consider user/external aspects when drawing constraints (which are derived from use cases, system requirements, etc.); then he/she needs to hold the programmer/internal perspective to construct the mental model and simulate it; then again, he/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 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. This leads the programmer to hold a 'usability-oriented' perspective, and it is consistent with our findings on LSC.

4 Summary and conclusions

We have studied the question of whether working with the language of *live sequence charts* (LSC) leads programmers to adopt a more user-oriented perspective. Our findings demonstrate that when working with LSC the students' perspective was more user-oriented than their perspective when working with conventional programming means. In way of 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 programmers to concentrate more on the former.

From this result, we derive the following conclusions:
i) Reducing context switching has a positive effect on performance, and is especially desired in the case of novices. Hence, we believe that the main concepts of LSC are very suitable for novice programmers.
ii) Holding a more user-oriented perspective can help programmers create more usable artifacts whose specification is more complete and better defined. Our data showed some indication of this, but it requires further study.
iii) In the context of end-users, combining user perspective with the ability to apply domain knowledge directly, through scenario-based and direct interface programming, renders the LSC approach especially useful in the domain of end-user programming.

5 Acknowledgments

The first-listed author would like to thank Moti Ben-Ari and Rivka Taub for useful discussions.

References

1. Beth Adelson and Elliot Soloway. The role of domain experience in software design. *IEEE Trans. Software Eng.*, pages 1351–1360, 1985.
2. Mordechai Ben-Ari and Tzippora Yeshno. Conceptual models of software artifacts. *Interact. Comput.*, 18:1336–1350, December 2006.
3. Michelene T. H. Chi. Quantifying qualitative analyses of verbal data: A practical guide. *Journal of the Learning Sciences*, 6(3):271–315, 1997.
4. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.
5. Michael Eisenberg. Programmable applications: interpreter meets interface. *ACM Sigchi Bulletin*, 1995.
6. David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987.
7. David Harel. From play-in scenarios to code: An achievable dream. In Tom Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 22–34. Springer Berlin / Heidelberg, 2000.
8. David Harel and Michal Gordon-Kiwkowitz. On teaching visual formalisms. *IEEE Softw.*, 26:87–95, May 2009.
9. David Harel and Rami Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and System Modeling (SoSyM)*, 2:2003, 2002.
10. David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
11. David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
12. Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
13. Bonnie A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1993.
14. Donald A. Norman. Cognitive artifacts. In *Designing interaction*, pages 17–38. Cambridge University Press, New York, NY, USA, 1991.
15. Amir Pnueli. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:46–57, 1977.
16. Alan H. Schoenfeld. *Mathematical problem solving*. Academic Press, Orlando, FL, 1985.
17. Antony Tang, Aldeida Aleti, Janet Burge, and Hans van Vliet. What makes software design effective? *Design Studies*, 31(6):614 – 640, 2010. Special Issue Studying Professional Software Design.