On Teaching Visual Formalisms

David Harel and Michal Gordon-Kiwkowitz, *Weizmann Institute of Science*

A graduate course on visual formalisms for reactive systems emphasized using such languages not only for specification and requirements but also (and predominantly) for actual execution.

In the spring semester of the 2004–2005 academic year at the Weizmann Institute of Science, coauthor David Harel delivered the graduate course Executable Visual Languages for System Development.

As with most courses at our institute, it was mainly for MSc students, but several PhD students participated too. About 30 students participated. Most had an undergraduate degree in computer science. Some had bioinformatics degrees, and a few had undergraduate training in biology and were performing biological-modeling research. Coauthor Michal Gordon-Kiwkowitz was one of the students.

Although other courses teach similar subjects (for more information, see the "Relation to Other Courses" sidebar on page 90), as far as we know, none concentrate on teaching visual formalisms as executable programs. Our aim was to convince students that you can use some visual formalisms to program complex reactive systems and that support tools exist that can serve as conventional programming environments. Using those formalisms and tools, you can program systems intuitively and graphically. You can then execute the resulting visual artifacts, analyze them, and use them to automatically generate a final implementation.

Two Programming Paradigms

The course was based on two programming paradigms. The first is the *intra-object* approach, which is the accepted way of programming a system. In this approach, after identifying the system's structure—that is, its objects and their interrelationships—you program each object's behavior separately, usually in a state-based fashion. An implementation then consists of the collection of each object's behaviors. In the course, we used object/class diagrams for the structure, statecharts for behavior,[1,2] and Rhapsody (www.telelogic.com/products/rhapsody/index.cfm) as the supporting tool.

The second paradigm is the *interobject* approach. As in the intra-object approach, you first identify the system's structure, including the objects' local abilities (for example, methods and operations). However, you then program behavior through multimodal scenarios specifying interobject behaviors. In this approach, we used live sequence charts (LSCs)[3] for behavior and the Play-Engine[4] as the supporting tool.

In line with this, the students built two related systems of their own choosing, using the intra-object approach for one and the interobject approach for the other. They then combined the two systems using the InterPlay tool.[5] In many courses, students must implement predefined projects (with a main point of interest being the

0740-7459/09/$25.00

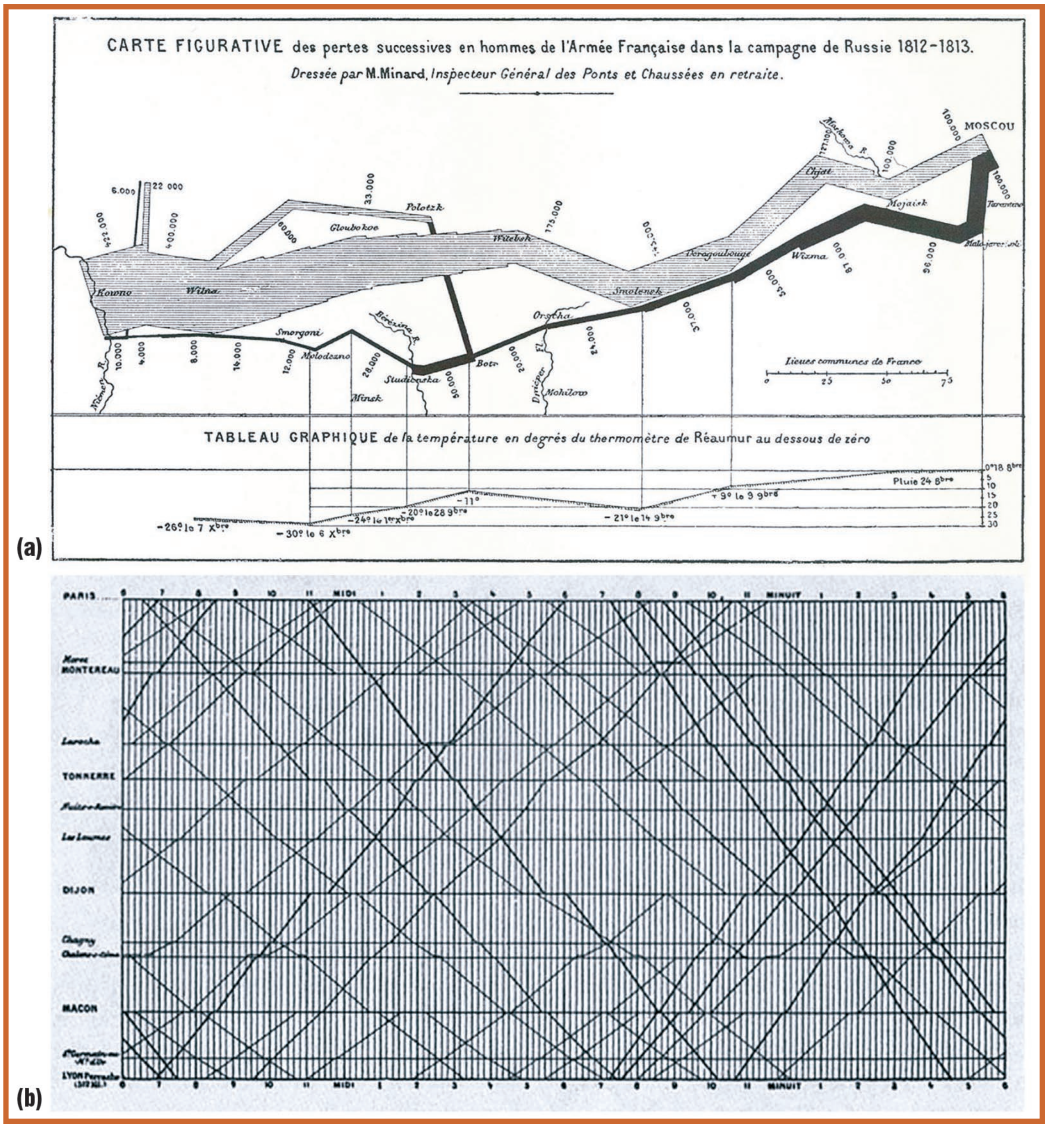


Figure 1. Two early examples of the visual specification of time-dependent data: (a) an 1861 graph depicting Napoleon's march to Moscow and retreat to Kovno and (b) an 1880 depiction of the 1823 railroad schedule and train dynamics between Paris and Lyon.[8]

particular tools and methods that the students choose).[6,7] However, we were particularly interested in what types of systems the students would choose when constrained by the requirement to work with the specified visual languages.

Motivation and Introductory Material

The course started with two lectures on visuality and capturing information visually. We emphasized that pictures are mostly static and that a main challenge of visuality is specifying and capturing dynamics. A quote from "A Map of Verona," a 1942 poem by Henry Reed, came in handy:

Maps are of place, not time, nor can they say
The surprising height and colour of a
building,
Nor where the groups of people bar the way.

So maps, and by extension most visual aids, are about static artifacts (place), not dynamic ones (time). They can't or aren't intended to capture time-dependent occurrences, such as groups of people barring the way. We then showed examples of using visuality to capture complex data—for example, bar graphs and zoomable descriptions of election results.

However, in line with the desire to emphasize dynamics and to deal with the visual specification of time-dependent data, we then showed some extraordinary early examples featured in Edward Tufte's books.[8–10] These included Charles Joseph Minard's famous 1861 illustration of Napoleon's march to Moscow and retreat to Kovno (see Figure 1a) and the 1880 depiction of the entire 1823 railroad schedule and train dynamics between Paris and Lyon in a single grid-like diagram[8] (see Figure 1b).

We then briefly discussed standard visualization techniques for computer programs, particularly flowcharts. By and large, we claimed, flowcharts have been a failure. They're used to help visualize algorithmic processes, but the emphasis here is on "help." They have never been widely accepted as part of a true programming medium. The same goes for some attempts to devise visual programming languages, which are often 2D schemes for laying out icons, with the dynamics derived from the location and proximity thereof. They're somewhat like a 2D version of APL. In the past, this had led some people to dismiss visual languages completely. (Edsger Dijkstra and Frederick Brooks were among the most outspoken opponents of diagrammatic methods and visuality in programming and software engineering.)

Reactive Systems and Visual Formalisms

The course then went on to discuss reactive systems[11] and the acute difficulty of capturing their behavior. For transformational systems, a structural or functional decomposition and careful data-flow analysis yield a temporal, dynamic, executable model of the system. However, specifying such a decomposition for a reactive system says nothing much about the dynamics. So, specifying reactivity is much more difficult.

Basically, reactivity constitutes the discrete part of dynamics. Reactive systems include as special cases most types of concurrent, distributed, computer-embedded, and real-time systems. They appear in numerous application areas: automotive, aerospace, interactive software, telecommunications, hardware, medical instrumentation, control systems, and so on. Moreover, it's becoming apparent that many kinds of natural systems, such as in biology and physics, and systems in the social and economic sciences and healthcare are also reactive.

Obviously, the choice of reactive systems as the course's subject was influenced by the instructor's particular research interests. However, more objectively, it's justified by the wide agreement that reactive systems' behavior is the most difficult aspect of system specification and implementation. Reactive behavior is a crucial, often absolutely critical, aspect of many kinds of computerized systems. It exists both between the system and its environment and among the system's various parts. Reactivity complicates everything in system development: requirements, specification, design, implementation, verification, maintenance, and so on. And although reactivity centers on the discrete, we argued that continuous aspects can be crucial too, leading to the notion of a hybrid system.[12]

To strengthen the point about dynamics, we drew analogies to transportation. Houses and bridges are there to *be*, whereas software and systems—and cars, trains, and aircraft—are there to *do*. Methods and tools that support only the structural and functional aspects of system design are good for nonreactive systems. However, for systems with strong reactive dynamics, they're like building a car with no engine.

Sadly, the system development tools offered to engineers in the 1970s and much of the 1980s (the so-called CASE tools) were really engineless cars. They were nice cars—with leather upholstery, great suspension, and electronic everything—but they could go nowhere. CASE tools could do excellent graph editing, documentation, requirements capture and analysis, project management, and more. However, they couldn't execute the models you built with them and didn't generate full runnable code. So, you had to separately write the crucial parts of the actual code. Imagine a programming-language environment that has everything you could dream of to help you program, except a compiler (or a running interpreter).

So, the problem is to devise a framework for developing complex reactive systems, consisting of means for describing and analyzing the structure but driven and propelled by the behavior. The underlying tools ought to enable full executability of the models and should automatically generate running code.

We then talked in detail about the concept of a visual formalism,[13] especially how you would apply it to specify reactive behavior. We chose the term carefully, so that the first word hints at the desire for visuality, intuition, and clarity and the second hints at the need for mathematical rigor. This isn't about pretty pictures or doodling but about real, formal, rigorous programming of a system visually. We discussed that visuality here means using topology as much as possible (connectivity, "insideness," disjointedness, open versus closed curves, and so on) and then using geometry only if needed (size, shape, line style, color, and so on). Icons can be used too, but sparingly—we don't want an iconic language.

We then discussed formality, stating that a formal specification isn't necessarily a formal-looking specification. A language doesn't need a lot of Greek letters to be formal. This led to a careful discussion of syntax, semantics, and their relationship, and the special difficulties these raise when the languages are visual—and thus not necessarily compositional.[14]

Relation to Other Courses

As we mentioned in the main article, our course's main objective was to teach executable visual formalisms to graduate computer science students and researchers. Other courses most relevant to ours teach either reactive systems, in which case they typically focus on real-time methods and system development, or UML, in which case they usually concentrate on design principles. Our course took neither path, so a comparison is difficult.

Courses on Real-Time Systems

Tal Lev-Ami and Shmuel Tyszberowitcz reported on a course that focused on developing provably correct reactive real-time systems.[1] Like our course, theirs combined theoretical issues with practical implementation. However, the theory focused on how to specify requirements and automatically verify the implementation against those requirements. The course taught models of both reactive and real-time systems; concepts, methods, and tools for specification; analysis and design of real-time systems; the synchronous model; verification methods; and scheduling algorithms. The students designed systems, used the Esterel language to implement those systems, and then used the Esterel simulator to analyze and check the systems' behavior. They later verified the system properties and specified assertions using various model checkers, such as SMV (Symbolic Model Verifier), TempEst, and XEVE (the Esterel Verification Environment).

Although we also required students to test their system using test cases, we assumed previous knowledge of verification methods and didn't explicitly teach them. Our point was the inherent connection between using a visual formalism—for example, sequence diagrams for implementation and testing, rather than verification per se.

Also, in Lev-Ami and Tyszberowitcz's course, the students selected their projects from a closed set of projects, including systems such as an answering machine, an elevator controller, a washing machine, and a home-alarm control module. These projects were selected to be appropriately small or medium-sized and assumed that most students were experienced with the analysis of large information systems.

In contrast, although our students were experienced graduate students as well, we let them select their own project, as long as it used the constructs they learned in the intra-object and interobject modeling approaches. (For more on these approaches, see the main article.) Our course focused more on contrasting these approaches and not on real-time issues, such as scheduling.

From a course on reactive and real-time systems similar to that of Lev-Ami and Tyszberowicz, Ran Lotenberg and Tyszberowitcz presented student projects that stressed the full life cycle of system development.[2] They described the projects in detail so that other instructors could use them. We describe our projects to provide a peek at the possibilities students see when they understand reactive systems and the visual tools for specifying and executing their behavior.

The Intra-object Approach

In this part of the course, we first described the basic elements of the language of statecharts.[1] This language is an extension of state transition diagrams that's enriched by a hierarchy of states and orthogonal (concurrent) components, with transitions allowed between and among all levels.

Statecharts

We reemphasized our point about topological constructs taking first-class status. We showed how connectedness, insideness, detachment, intersection, and so on are central to statecharts. Also, they can be visualized easily, and the brain recognizes them immediately.

Next, we described how to link statecharts to a system's structure. We chose the object-oriented (OO) approach (which was really basic to the entire course). So, we described the language of class diagrams and object model diagrams (OMDs), as in UML.[15] We also discussed OO statecharts in detail, including their operational semantics[2,16] and emphasizing their full executability. We then described and demonstrated Rhapsody.

We repeatedly emphasized and explained the resulting models' intra-object nature. In this approach, you typically structure a model as an object system and then specify a behavior by assigning a statechart to each object. The entire collection of statecharts can then be executed or translated automatically into fully runnable code.

This central aspect of statechart models leads to two important points. First, in Rhapsody, the model isn't executed directly. Rather, the automatically generated code is executed, and the charts are continuously animated using the execution's feedback. (In contrast, in Statemate, Rhapsody's non-OO precursor, statecharts could be executed directly without code generation.) Second, the events, conditions, and actions in a statechart must be written in some language. A dispute exists about whether this should be a specially devised *action language* or a fragment of a standard programming language. Rhapsody takes the latter view; we based the course on a version of the tool in which C++ serves as both the language for writing the model's nongraphical parts and the target language for the code

Other real-time courses such as Wolfgang Halang's[3] are aimed at engineers and teach design and implementation of real-time systems, emphasizing such topics as task scheduling, hardware architecture, process interfacing, fault tolerance, and project integration. We didn't deal with such issues (for example, we ignored scheduling issues unless the modeler wanted to explicitly program them into the behavior). Many approaches to real-time systems stress determinism; our course presented the view that a reactive system can also achieve its goal if it's nondeterministic or if the determinism isn't specified directly but emerges from the implementation. Halang suggests implementing systems using the real-time language Pearl and concentrating on its real-time features. He also discusses selecting operating systems for real time, suggesting that students implement a small real-time operating system, which wasn't in our course's scope.

UML Courses

Because our course concentrated on modeling techniques, specifically statecharts and live sequence charts, you could compare it with courses that teach UML to software engineers. Gregor Engels and his colleagues' course taught software engineering concepts, such as requirements specification, analysis, and design, through teaching UML.[4] They claimed that through UML, students can become familiar with more abstract concepts and design principles.

The two main visual formalisms our course dealt with are central to UML. Statecharts heavily influenced UML's conception from its early stages and are its main behavioral language. The more recent live sequence charts influenced the move from the message-sequence-chart-like diagrams of earlier versions of UML to those of UML 2.0. We didn't teach UML as simply a means for design but stressed that you can use some visual formalisms for both design and implementation.

Other UML courses, such as that of Dirk Frosch-Wilke,[5] teach students how to use the diagram notation they're familiar with from system analysis and design for industrial-scale projects. We found no courses that focus on teaching experienced students to program full projects using visual formalisms, especially not when behavior is crucial.

References

1. T. Lev-Ami and S. Tyszberowitcz, "Reactive and Real-Time Systems Course: How to Get the Most Out of It," *Real-Time Systems*, vol. 25, nos. 2–3, 2003, pp. 231–253.
2. R. Lotenberg and S. Tyszberowitcz, "Student Projects in Reactive and Real-Time Systems Course," *Proc. 3rd IEEE Real-Time Systems Education Workshop*, IEEE CS Press, 1998, pp. 57–62.
3. W.A. Halang, "A Curriculum for Real-Time Computer and Control Systems Engineering," *IEEE Trans. Education*, vol. 33, no. 2, 1990, pp. 171–178.
4. G. Engels et al., "Teaching UML Is Teaching Software Engineering Is Teaching Abstraction," *Proc. MoDELS 2005 Workshops*, LNCS 3844, Springer, 2005, pp. 306–319.
5. D. Frosch-Wilke, "Using UML in Software Requirements Analysis—Experiences from Practical Student Project Work," *Proc. Informing Science + IT Education Conf.*, Informing Science Inst., 2003, pp. 175–183.

generator. (In contrast, Statemate has its own action language.)

We then described reactive animation, whereby you can use a state-of-the-art reactive-system tool, such as Rhapsody, linked directly and smoothly with an animation tool, such as Flash or 3D Studio.[17] The goal is to specify systems for which the highly dynamic front end requires more than a GUI—namely, true animation. We illustrated this by a biological model of T cell development in the thymus.[18]

The First Project

About halfway through the course, we gave pairs of students one month to model a small reactive system. They built a specification with OMDs and statecharts and implemented it using Rhapsody. The basic guideline was to make the example somewhat complex. The model should have a reasonably nontrivial number of states, it should contain orthogonality and other basic features of the statechart language, and so on.

The spectrum of systems they chose was extremely interesting. Some chose the kind of electronic or technical examples that often illustrate reactive systems, such as an MP3 player, an air conditioner, and a car radio. Several chose far less standard systems, such as a towing company, a family that owns a falafel store, a combinatorial bit-shifting game, and—perhaps the most surprising—a system that plays the darbuka (a Middle Eastern drum), rhythm changes and all. (For more on the projects, see the supplement to this article at www.computer.org/software/webextra.html)

The Interobject Approach

In this part of the course, we first discussed the language of *message sequence charts* (MSCs)[19] (or the UML version, UML sequence diagrams[15]), demonstrating its inadequacy for specifying behavior. This language is normally for testing: you set down some scenarios as requirements and then test that each scenario is satisfied by some appropriate run of the system. To specify the system model, you use conventional intra-object methods. However, an MSC contains no prescriptive semantics: it gives only a possible set of occurrences and a partial order on them.

A formal specification isn't necessarily a formal-looking specification.

Live Sequence Charts

Next, we introduced LSCs, which are an enrichment of MSCs with universal and existential (hot and cold) modalities. So, the LSC language can express mandatory scenarios, conditional scenarios, possible scenarios, forbidden scenarios, and many other such combinations. Its full version allows time specifications, symbolic instances, probabilistic choice, and more.[4] As in the intra-object part of the course, we described the language and its features, including how LSCs deal with objects and their internal capabilities, such as methods and operations.

We then discussed the two main methods for working with LSCs. In *play-in*, you specify the behavior directly through the system GUI, and the supporting tool constructs the LSCs automatically on the fly. *Play-out* is the rather subtle execution mechanism of LSCs, which does what the scenarios prescribe but without ever causing violations.[4] Here too, we described the language's operational semantics in detail. We then introduced and illustrated the Play-Engine,[4] which supports LSCs and the two play methods.

Besides presenting the "standard" examples, we presented a nontrivial biological system modeled using LSCs: fate determination of the vulval precursor cells in the *C. elegans* nematode.[20]

The course then discussed InterPlay, a recent tool that lets you connect interobject and intra-object specification.[5] InterPlay is a simple mechanism for linking models—for example, a Rhapsody model with a Play-Engine model—such that the objects you specify using statecharts are considered external to the LSC specification and the Play-Engine. So, both tools can run in tandem, with the Play-Engine graciously handing over to Rhapsody whenever it awaits a response from a statechart-modeled object.

The Second Project

The students then used LSCs and the Play-Engine to model a system that was an extension of their first system and used InterPlay to link them. If this turned out to be unnatural, students could add a small number of objects to one of the systems and then link the two parts. In short, the students had one month to deliver a fully executable two-part system—an intra-object part specified with statecharts and an interobject part specified with LSCs. Figure 2 shows a statechart from the Tow Companies project and an LSC that connects to the statechart simulator and generates a similar behavior.

A Student's Viewpoint

At the course's end, the students filled out a questionnaire regarding their opinions about the two approaches, the tools, and the assignments. Here we present some of the comments, integrated with Michal Gordon-Kiwkowitz's opinions.

A Paradigm Change

The course was a sort of microcosm that tried to convince moderately experienced programmers that visual languages are a real possibility. One question that comes to mind after learning about visual languages and the intra-object and inter-object approaches is, can visual languages be the next generation of programming?

One barrier to the acceptance of visual languages is their difference from traditional ones. When you're familiar with one textual programming language, learning the new syntax for another textual language is simple. Consider, for instance, learning VB.NET after programming in C++. The main methodology is known: start by writing some kind of main function; create your classes, variables, and functions; and so on. Only some of the syntax changes.

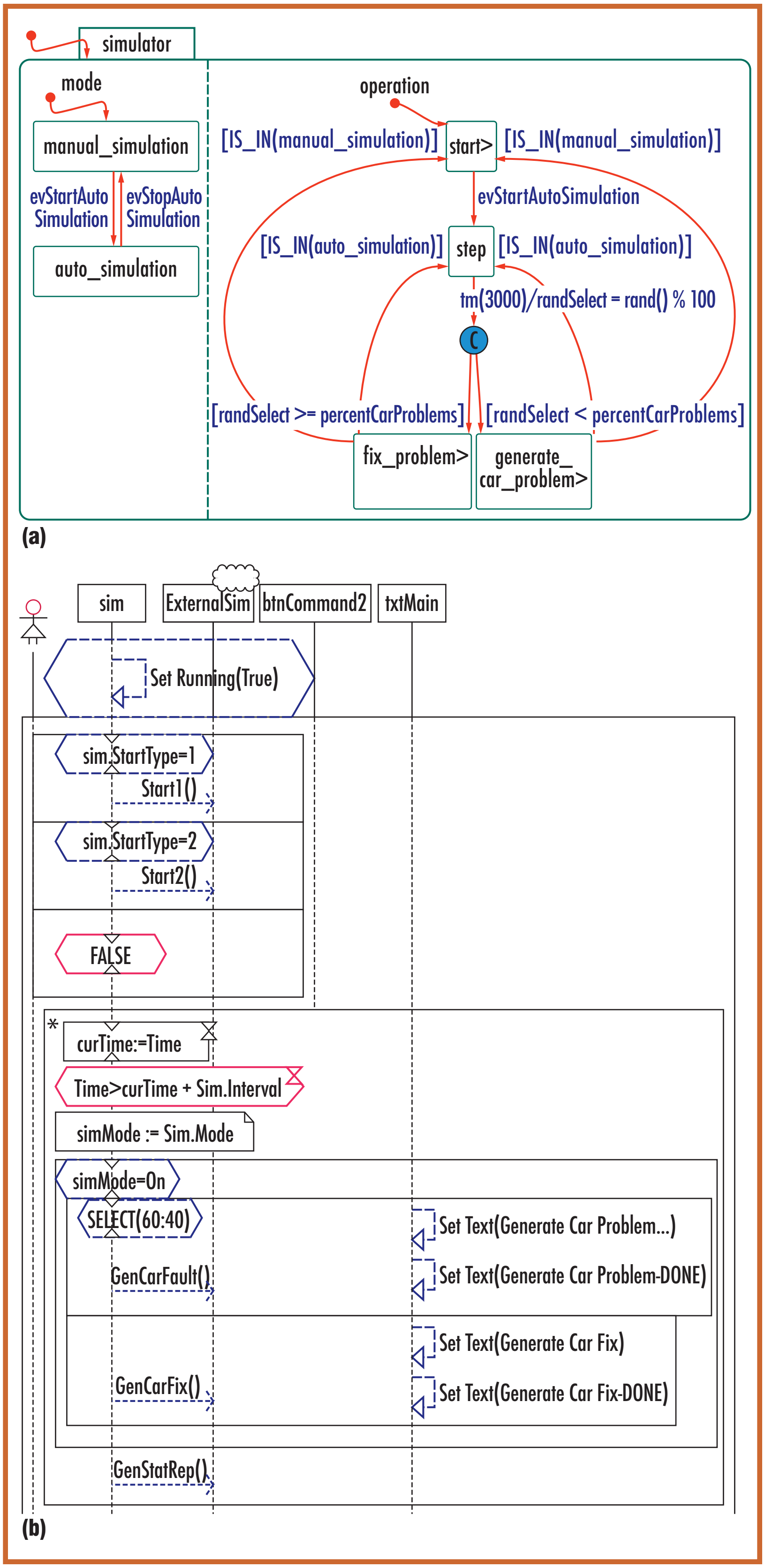
In contrast, when starting to program using visual languages, you must learn both a new syntax and a new methodology. It's easy to feel lost; this is especially true for experienced programmers who already "speak" a textual programming language. Many students commented that using languages they knew would have been easier. Drawing states to describe an object's behavior and using play-in to specify the behavior are intriguing approaches that are applicable to many software development stages. However, the methodology should be modified to help programmers assimilate the very different tools of visual languages.

Another barrier is that visual languages aren't prominent, either in industry or schools. For visual programming to be more common, teaching software development must change considerably. This can be done, for example, by enriching UML courses with discussions of executability and its crucial importance, or by discussing visual executable languages when introducing students to programming.

Statecharts and Rhapsody

During system planning, programmers often intuitively draw statecharts and the like to clarify the system to themselves and the different development teams. They also commonly use finite state diagrams to define object behavior. So, the statechart language is intuitive, making execut-

Figure 2. Linking the intra-object and interobject approaches: (a) a statechart of the towing-company simulator, which generates random faults to drive the simulation, and (b) a live sequence chart for automatic generation of random faults for the towing-company simulator. The LSC is connected to the statecharts simulator using an external object. The students used InterPlay to execute the systems in tandem.



able programming more like developers' natural thought processes.

However, students "speaking" fluent OO programming languages found programming using statecharts more difficult than did less experienced programmers. Even those who had already encountered concepts similar to statecharts or worked with UML had to come to grips with the language's innovations: the rigid formalism, the hierarchical nature, and using Rhapsody to execute a program.

When students built the model for the first project, having an OO education in formal languages hindered the initial development process. The natural, but incorrect, flow when programming was to create some state with large blocks of code that handled that state's behavior, rather than create several states to describe the behavior. Initially, students had to pay special attention to ensure that their system included enough states to satisfy the assignment's spirit. In many cases, students had to reform blocks of code into states and transitions, gradually transforming a code-based model into a state-based one.

Because Rhapsody also lets you write textual code, some students found using it more natural than using statecharts. Several students commented that statecharts are a nice idea but that traditional programming is much easier. It's tricky to isolate how much of this feeling comes from previous familiarity and experience with textual languages and how much is truly related to the statechart language. Although knowledge of one textual language helps you learn others, it might interfere with acquiring the skills for a totally different approach (something that's also true for learning natural languages with different grammar rules).

If you're skeptical about visual programming languages presenting a barrier to programmers familiar with textual languages, consider programming's history. For programmers skilled with punch cards, writing code must have seemed quite peculiar, but few people would argue that this was a detrimental revolution. Similarly,

One lesson from all this is that the audience is an important factor in teaching visual languages.

assembler programmers resisted the transfer to high-level languages, believing that they would have less control over the system's behavior and less freedom of choice.

Students taking this course sometimes had a similar feeling that statecharts were less powerful than textual languages, although theoretically they have the same power. So, when students had the chance to revert to conventional code, they often preferred it, especially at the course's beginning.

Not surprisingly, learning takes time. As the course advanced, the students became more comfortable with the language and tool and created more states and less textual code. Ultimately, using statecharts and Rhapsody, the students built systems that worked and were easy to modify, debug, and especially review.

One lesson from all this is that the audience is an important factor in teaching visual languages. Teaching this subject to students familiar with some standard textual language might be harder than teaching it to nonprogrammers; it would be interesting to compare these two groups' learning curves. On the other hand, for someone with no textual-language experience, the course would probably be a challenge. This is because Rhapsody was designed from the viewpoint of textual OO programming, and some textual programming is almost inevitable when constructing objects, initializing arrays, and so on.

LSCs and the Play-Engine

Although the interobject approach was new to most students, they learned it more quickly. This might be because what the students modeled mainly involved interaction with the system's interface, not the system itself or its logic. It might also be because the LSC language doesn't enable the use of low-level textual code as Rhapsody does, which decreases the potentially bad effects of the interference of old habits. With a totally new approach, which is distinctly different from textual programming, and no ability to revert to the textual tongue, the learning process was easier than with the intraobject approach.

You could argue that the students' programs don't provide solid evidence of the interobject approach's viability because they were mostly interfaces to a different system. However, many applications today are just that, and the LSC and play-in/play-out approach is an agile, intuitive way to build such applications.

Many students mentioned that although connecting a GUI to the Play-Engine environment by specifying the behavior through play-in was cumbersome (a technical problem that can be easily overcome in a commercial tool), the approach was remarkable. It's precisely how designers think when writing specifications. Because the students chose their own project, they were in essence designing their own interface and system. In this case, the planning and programming are very similar: "when the user presses the X button, close the monitoring window," or "when the user enters a message in the text box and clicks the OK button, display the message in the status bar." So, the specification and programming occur simultaneously. This should save many person-hours because the shift from specification to application is smoother.

The students had much to say about the Play-Engine's current (academic, noncommercial) version. For example, it doesn't seem suited for large-scale projects. However, a leap of faith and a long-term outlook are necessary for changing how systems are developed. A large-scale system in any language requires breaking the project into modules, and such a system will be difficult to track and manage in any language. The play-in/play-out approach, if adopted, will probably evolve to deal with larger projects.

Perhaps this approach needs time to mature, but it has advantages over known textual languages. First, instead of humans describing their request in the computer's language, they can specify what they want almost in their own natural language—showing, pressing, and doing. This prevents many mistakes that emerge from the gap between the system designer and the application programmer. Second, this approach uses as essential programming tools the mouse and the graphics, which weren't available when textual languages were developed. These tools are powerful and appealing to today's generation. Seeing the application in operation while you're designing it makes design easier. It can also help avoid problems that normally would be evident only after development, which is useful for agile system development.

Programming for Soft Tasks

Consider system engineers and business management and accounting students. Not having much knowledge in computation and mathematics, they usually take some Visual Basic courses. So, they should be able to easily build useful "soft" applications—light, agile applications for specific tasks, customized for a specific job. However, after handing in the course exercises, not many of them ever write anything in Visual Basic again.

You might wonder whether these people—who are new to programming and don't wish to know about it in depth—would be more productive had they learned a more appealing language. The intuitiveness of LSCs and the play-in/play-out approach might be attractive to them. The question remains, how would nonprogrammers react to tools that manipulate visual languages that are closer to their normal planning and thought processes?

Overall, our experience with the course was positive. The students gave much constructive feedback about the languages and tools. We've already applied some of their comments to the interobject approach and the relatively new (and still insufficiently tested and solidified) Play-Engine.

One of the most interesting things this course demonstrated was the variety of applications for which these programming languages are appropriate, even when used by students with only a few weeks' experience with the languages. This supports David Harel's long-held belief that visual formalisms represent a useful, intuitive way to program systems, not just specify their desired behavior in the requirements or specification phases.

As to how best to teach visual formalisms, this course was only an initial feasibility test; the jury is still out.

Acknowledgments

We're grateful to the students who agreed to have their projects featured in the appendices and then rewrote them to fit the article's style and format: Yishai Admanit, Mica Arie-Nachimson, Yoram Atir, Erez Kantor, Guy Nachimson, Avital Sadot, Tal Shay, and Gera Weiss. PhD students Shahar Maoz and Yaki Setty, the official teaching assistants, not only helped design the course and grade the homework but also delivered lectures. Na'aman Kam gave a lecture about the *C. elegans* project. Dan Barak was indispensable as our systems expert, and his presence was especially crucial for the InterPlay parts. Michal Gordon-Kiwkowitz's research was supported partly by the Weizmann Institute's John von Neumann Center for the Development of Reactive Systems.

References

1. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, 1987, pp. 231–274.
2. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Computer*, vol. 30, no. 7, 1997, pp. 31–42.
3. W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, 2001, pp. 45–80.
4. D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer, 2003.
5. D. Barak, D. Harel, and R. Marelly, "InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming," *IEEE Trans. Software Eng.*, vol. 32, no. 7, 2006, pp. 467–485.
6. T. Lev-Ami and S. Tyszberowitcz, "Reactive and Real-Time Systems Course: How to Get the Most Out of It," *Real-Time Systems*, vol. 25, nos. 2–3, 2003, pp. 231–253.
7. R. Lotenberg and S. Tyszberowitcz, "Student Projects in Reactive and Real-Time Systems Course," *Proc. 3rd IEEE Real-Time Systems Education Workshop*, IEEE CS Press, 1998, pp. 57–62.
8. E.R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, 1986.
9. E.R. Tufte, *Envisioning Information*, Graphics Press, 1990.
10. E.R. Tufte, *Visual Explanations: Images and Quantities, Evidence and Narrative*, Graphics Press, 1997.
11. D. Harel and A. Pnueli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems*, K.R. Apt, ed., Springer, 1985, pp. 477–498.
12. O. Maler and A. Pnueli, eds., *Hybrid Systems: Computation and Control*, LNCS 2623, Springer, 2003.
13. D. Harel, "On Visual Formalisms," *Comm. ACM*, vol. 31, no. 5, 1988, pp. 514–530.
14. D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of 'Semantics'?" *Computer*, vol. 37, no. 10, 2004, pp. 64–72.
15. "UML Resource Page," Object Management Group, www.uml.org.
16. D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)," *Proc. Integration of Software Specification Techniques for Applications in Eng.*, LNCS 3147, Springer, 2004, pp. 325–354.
17. S. Efroni, D. Harel, and I.R. Cohen, "Reactive Animation: Realistic Modeling of Complex Dynamic Systems," *Computer*, vol. 38, no. 1, 2005, pp. 38–47.
18. S. Efroni, D. Harel, and I.R. Cohen, "Towards Rigorous Comprehension of Biological Complexity: Modeling, Execution and Visualization of Thymic T Cell Maturation," *Genome Research*, vol. 13, no. 11, 2003, pp. 2485–2497.
19. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, Int'l Telecommunication Union, 1996.
20. N. Kam et al., "Formal Modeling of *C. elegans* Development: A Scenario-Based Approach," *Proc. 1st Int'l Workshop Computational Methods in Systems Biology* (ICMSB), LNCS 2602, Springer, 2003, pp. 4–20.

About the Authors

David Harel is the William Sussman Professor in the Weizmann Institute of Science's Department of Computer Science and Applied Mathematics. His research interests are software and systems engineering, modeling biological systems, and the synthesis and communication of smell. He's the inventor of statecharts and co-inventor of live sequence charts, and he codesigned Statemate, Rhapsody, and the Play-Engine. Harel has a PhD in computer science from MIT. He has received the ACM Karlstrom Outstanding Educator Award, the Israel Prize, and the ACM Software System Award. He's a fellow of the ACM, IEEE, and AAAS. Contact him at dharel@weizmann.ac.il.

Michal Gordon-Kiwkowitz is a PhD student in the Weizmann Institute of Science's Computer Science and Applied Mathematics Department. She's working on intelligent human interfaces for programming using the language of live sequence charts. Her research interests include intelligent human interfaces, visual programming languages, software design, cognitive psychology, and human and computer vision. Gordon-Kiwkowitz has an MSc in computer science from the Weizmann Institute of Science. Contact her at michal.gordon@weizmann.ac.il.