Can Programming Be Liberated, Period?

David Harel
Weizmann Institute of Science

The author describes his dream about freeing ourselves from the straightjackets of programming, making the process of getting computers to do what we want intuitive, natural, and also fun. He recommends harnessing the great power of computing and transforming a natural and almost playful means of programming so that it becomes fully operational and machine-doable.

Nine years ago, I sat down to write about a dream, one that would allow us to go from intuitively "played-in" scenarios to running code. Some of its most technically challenging parts were stated without providing too much support for their feasibility. Hence the choice of the term "dream." Ever since that paper was first published in 2000,[1] not only hasn't the dream evaporated, but it has continued to have a nagging presence, looming even larger in my mind, and getting broader and more elaborate by the year.

More significant is the fact that quite a bit of work has been carried out since then, which, while still a far cry from justifying the replacement of a dream by a plan, does now seem to offer some preliminary evidence of feasibility. Consequently, I've decided to revisit the topic and to describe the dream anew, or, more correctly (but possibly not very wisely), to propose a more dramatic and sweeping version thereof.

I should apologize to the reader at the start that this article doesn't get very specific or technical at all. Moreover, with the exception of the sidebar, it might read like the ramblings of a crazed, or dazed, individual. I should also point out that this article's title is, of course, intended to be a catchy take on the title of John Backus's wonderful Turing Award lecture and paper, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs."[2]

PROGRAMMING'S STRAIGHTJACKETS

We've come a long way since programming had to be done by tediously listing machine-level instructions that prescribed how a specific computer was to modify and move bits and words in its memory. It's not my intention to attempt a survey of the history of programming. Still, it's obvious that there has been an amazing transition up the language-generation ladder, from machine languages to assembly languages, then to conventional imperative programming languages, and from there to the variety of contemporary programming styles—functional, logical, concurrent, visual, synchronous, constraint, object-oriented, aspect-oriented, and on and on. And there also have been numerous special-purpose languages, constructed for specific kinds of applications.

However, there is a sense in which programming is still the same kind of technically tedious task, albeit carried out on a higher, more appropriate, level of abstraction. It still entails writing programs, usually by using symbols, keywords, and operational instructions to tell the computer what we want it to do. A compiler is but a means to translate in reverse, down the generation ladder, rendering high-level programs readable and executable by the machine. And programming still requires testing and debugging, or preferably verification, to make sure that what we told our computer to do will have the results we desire. Moreover, we also must carefully specify the very

 Published by the IEEE Computer Society 0018-9162/08/$25.00 © 2008 IEEE

concept of what we desire, which often is as complex and as error-prone as the program itself.

In the present time and age, there is an additional complication, which stems from the ever-increasing number of applications that involve multiple "pieces" that operate concurrently and are often widely distributed. Just think of Internet applications and web services, for example. For these, the added issue involves having to program each part separately, and then having to make sure that the compound behavior, the orchestration of the parts, results in what we want.

I submit that, by and large, even the most modern approaches to programming still suffer from these constraints, which we might term the "three straightjackets of programming":

- (I) the need to write down a program as a symbolic, textual, or graphical artifact;
- (II) the need to specify requirements (the what) separately from the program (the how) and to pit one against the other; and
- (III) the need to structure behavior according to the system's structure, providing each piece or object with its full behavior.

Can we liberate programming from the keyboard, from having to specify both the what and the how, and from the system's structure?

Can we liberate programming from these three constraints: from the keyboard, from the thankless tension between the what and the how, and from having to partition the dynamics along the lines of the structure?

But what is the alternative, we might ask. How can we program a computer without telling it exactly what to do, and without having to use a tangible medium to inscribe that telling? How can we ever be sure that what we get is what we wanted unless we state both and then compare them? And how can we program a multitude of things, other than by giving each thing its own instructions?

Of course, there are entire approaches to programming for which the explicit intent is to remove or alleviate some of these constraints. Thus, for (I) above, researchers have developed novel techniques in certain specific application areas to decrease the effort involved in writing the relevant programs; query-by-example for relational databases is such a technique,[3] as are spreadsheets.

For (II), logical and functional languages,[2,4] as well as many attempts at what has been generically called automatic programming, such as the particularly ambitious and interesting idea of intentional programming,[5] are all intended to allow us to state what we want, with much of the how-it's-done left to the compiler or interpreter. The intent is similar for constraint-based languages[6] and special-purpose application generators. And for (III), the recent wave of aspect-orientation[7] tries to ease the need to totally align behavior with structure by making it possible to supplement the internal behaviors of objects with special kinds of cross-cutting behaviors that weave through several of them.

My dream is a lot more ambitious. I am much greedier, and want it all. Can we not, I ask, push the envelope on all of these, and in the process try to change the face of programming? Not just by a new generation of higher-level languages or an innovative methodology, but by approaching programming quite differently. My dream is about freeing ourselves from the straightjackets of programming, doing our work in a far more liberated way, making the process of getting computers to do what we want intuitive, natural, and also fun. I move that we harness the great power of computing to help in this very quest, in making a far more profound downward transition than that embodied in compilation, taking a natural and almost playful means of programming and transforming it so that it becomes fully operational and machine-doable.

And lest you are thinking, "Okay, here comes another one of those people—a guy with a magical solution to all problems," I should add: No, I don't have a solution. I don't have anything that works for all kinds of programming, and I definitely have nothing that is magical. However, some preliminary evidence indicates that the yellow brick road might be worth a walk, at least for a certain type of program.[8] But the message here is definitely "maybe we should be thinking about this some more," not "if you just do it my way you'll be fine."

THE DREAM, PART 1: PLAYING IN THE PROGRAM

Programming is not about doing; it's about *causing* the doing. We "program" all the time, although not necessarily computers. We get (or try to get) other people to do what we want, and we guide them to behave in ways we approve of. We bring up our children, we supervise underlings, and we run companies, departments, and faculties. We make sure (or try to make sure) that our stockbroker, our handyperson, and our real estate agent do what we want.

We achieve these things by issuing explicit instructions when needed. However, more importantly, this usually requires a combination of laying out general principles, showing or walking through examples of what we have in mind (or *being* an example), and prescribing rules and conditions for what can or must be done versus what must not or cannot be done. Increasingly, we rely on the accumulated abilities of the person being "programmed" to abide by this guidance (and of course on that person's willingness and integrity—or fear of the repercussions). To varying degrees, organizations and governments, as well as religions, also work that way,

getting people to be good citizens. Notice that restrictions and constraints are a natural and crucial part of this "programming." If it is important that something be done, no matter what, or that something is never done, we simply say so explicitly (or the book of regulations says so, or the Bible says so, or whatever), and the person doing the doing must comply.

My dream is to be able to program computers that way too. I'd like for us to be able to remove the double quotes from the previous paragraphs, to substitute computers and computational devices for people and organizations, and to find similar ways to make machines do what we want. Ways that come naturally to us and are a smooth extension of the way we think; ways that require far less technical prowess than today's programmers need, and which allow flexibility on the computer's part in achieving the goals we have set out while honoring our requirements and making sure not to violate any of our constraints.

So far, this sounds like an illusion—worse, a hallucination—rather than a good old solid dream. Let me try to put some flesh on it by talking about actual computers.

Although they are not quite like humans (notice the omission of the word "yet"), computers are coming along in leaps and bounds. It is hard to guess what the world of computing will look like in, say, 20 years, but we are already seeing amazing progress in language and voice recognition, vision, human-machine interfaces, logical and deductive abilities, heuristic reasoning, and much more. And all this without even mentioning the web and the way it's changing so many of our conceptions about computers and computing.

In many cases, the mathematics and algorithmics that underlie things computers do are getting more deeply buried inside, far from the user and often even from the programmer. And that's the way it should be, I claim, just like the way calculations underlying a spreadsheet are hidden from its user. In fact, the very borderline between user and programmer is becoming blurred.

Increasingly, computing calls for having to program incredibly complex *reactive systems*,[9] rather than systems whose role is to carry out numerous calculations. These are highly dynamic, discrete, event-driven systems, often with stringent timing constraints.

A reactive system's complexity is far less a result of complex computation and heavy algorithmics or the need to explore and mine intricate data. Rather, the system's complexity is a result of its subtle and complex (and often unpredictable) interactions with its environment and among the various parts of the system. These interactions consist of triggered and triggering events, changes in values, time-related constraints, probabilistic decisions, and so on, potentially happening in parallel in synchronous or asynchronous ways.

I believe that, as a general family of challenges for the world of computing, reactive systems are not only the hardest and most complex but also those in which centrality and significance will only increase. Again, no divine prophecy is required to see this; we only need to take an educated look at this Web-dominated, computers-are-everywhere era. Hence, although there are many significant kinds of nonreactive systems, and similar dreams might be articulated for them too, I dream about reactivity, to which the bulk of this article is devoted.

I claim that current methods for dealing with programming the dynamics of reactivity, however powerful and convenient, suffer from the same woes: We sit in front of a screen and write (or draw) programs that prescribe the behavior for each of the relevant parts of the system over time. Then we must check/test/verify that the combined behavior of all the parts satisfies a separately specified set of requirements or constraints. I dream of being able to do this quite differently.

Reactive systems are not only the hardest and most complex but also those in which centrality and significance will only increase.

Suppose you want to program a mobile phone. In fact, to make the story a little more direct, let's assume you're holding the phone in your hand, but that it's not a phone yet. It looks like one—it has a display screen, standard phone keys, four or five additional buttons, a port for communicating with the cellular antennas, and so on.

Let's further assume that you know the basic separate capabilities of each of these features. For example, the user can press and release a key, the device's internal illumination can be on, and so forth. The user can't manipulate the display, but the display can be on or off, show alphanumeric characters, display graphics and animations, and so on. However, other than knowing about these objects and their local capabilities, this is not yet a phone at all. It hasn't yet been endowed with phone-like behavior; you press a key, for example, and nothing happens.

What would be the most natural way to "teach" the device to be a phone? If you could talk to it, like you talk to a child or a student you are supervising, how would you proceed?

Well, as a start, you might say to it something like this: "Hey phone, whenever I press this key and hold it down for at least a half-second, you should switch on—meaning that your display should light up and show the cellular provider, my name, and the time." You might then give it similar instructions for switching off, possibly adding something like, "And, by the way, despite what you've just heard, don't ever switch off if the display shows that a text message is still in the process of being sent."

I should remark here that for the sake of this article, the example has been made rather simple. In actuality, we might also want to include in the switching on and off the starting and stopping of the communication between the phone's port and the cellular antenna. The protocols for these communication exchanges would have to be specified too.

You might next decide to start dealing with calls, saying something like, "Here's an example of how I'd like to make a call. If I press between three and 12 numeric keys sequentially, and then I press the green send key, you, in response, are to send out a call request to the cellular antenna containing the number formed by the pressed keys." To this you might add, "But if any two keys are depressed at the same time, you do nothing."

I'm not saying that this is *exactly* how I dream of programming a cell phone, but it's not that far off. Here are some relevant points.

Play-in is intended to constitute a natural and smooth computerization of how we'd cause some entity to behave the way we want.

First of all, if we don't have a phone, there's no point in trying to talk to it. Rather, we would be dealing with a computerized *mock-up image* of the phone, say, as a GUI on a computer screen. If we already have a graphical design for the phone, this would allow us to lay out the various objects as they would appear on the real thing; if not, we could show them in abstract form, say, as a structure diagram or object diagram. It would also make it easy to include in the process internal objects that the phone's user won't normally see, as well as nontangible objects that no one will normally see. Moreover, if we did have a real physical phone, but devoid of behavior, I can imagine working opposite it with no need for a soft mock-up of any kind.

Second, this process is about communicating to the system, in a natural style, the various pieces of behavior that we are interested in (or examples thereof) and teaching the system how to participate in them. These slices or chunks of behavior are not homogeneous in relation to the programmed system's desired overall behavior. They are *multimodal*; they can be specified to be a mandatory part of the behavior or a conditional part; they can be forbidden or preferred; they can be probabilistic, nondeterministic, or time-controlled; and so on. My point is that all of these are legitimate parts of the programming process—just like our telling someone what they can or cannot do, or when and under what conditions is part of our "programming" them.

Third, although advances in speech recognition and natural-language processing might eventually make it possible to freely talk to a phone or to its onscreen image, the sample session above is not about talking; it is about walking. That is, the process of realistically walking the system through the scenarios, hand-holding it while we show it, in a manner of speaking, how to cross a busy road without getting killed.

The term I have used for this in the past is *play-in*.[5] The point is not to talk about manipulating keys and displays, but to actually do the manipulation ourselves, in much the same way we expect to do it when the phone is built and we are *using* it rather than programming it. Rather than saying to the phone, "When I press this key …" the programmer would actually do it, for example, by clicking its onscreen image. Instead of saying, "You now make this light come on," we would actually show the phone what it should do by turning the light on—say, by selecting this action from a list of the light's capabilities.

Anything done in this way is done on the screen, or with the physical behavior-free system, in exactly the way we would like to see it done in the final programmed system. Representative examples, such as placing a call, would also be played in directly. The programmer would do this in a generic, by-example mode—just like the parent or educator—playing in an actual example of dialing a number, taking care to differentiate the example parts of the behavior from the fixed ones, and making the appropriate links—for example, the sample number dialed would have to be linked to the one sent out to the antenna.

Fourth, while many systems lend themselves nicely to GUI-based rendition, there is no a priori reason why we cannot carry out a similarly intelligent tutoring-like play-in process for systems whose front end is less discrete and less rigid. I am sure that experts on human-computer interaction would be able to come up with all sorts of analogs of the click, drag, and menu-select actions we do on GUI objects, which would work for playing in the behavior of more dynamically animated systems, such as games, navigation systems, automotive systems, tactical and avionics simulations, and so on. Thus, hybrid systems, which mix the discrete with the continuous and stochastic, are a special challenge here. Note, of course, that the better human-machine interfaces get, the richer play-in can become. A good example would be the ideas in Microsoft's experimental tabletop computing system.

Notice that I've said nothing yet about how to *run* "programs," only about how to "write" them. Nevertheless, it might make sense to pause here for a moment and see how this kind of intelligent play-in avoids the three main woes of programming—at least for the dynamics of reactive systems.

For issue (I), play-in is a walkthrough-style guiding, teaching, coaching, constraining process, playfully interactive, that is carried out directly and visually with the system's external or internal interface, and it does not

require the programmer to sit down and prepare a complete artifact of any formal kind.

For issue (II), both the what and the how are equally valuable parts of the play-in process, which can contain as much of either as the programmer decides. There is no need for separate specifications for the operational tasks and the requirements thereof. Anything that falls inside the total sum of what has been played-in will be a legal behavior of the system.

And finally, for issue (III) there is absolutely no requirement to capture or specify behavior per object/part/piece/chunk. On the contrary, I believe that the dynamic and interactive style of play-in encourages specifying interobject, or interpiece, behaviors whenever possible. But in any case, we should be free to specify behaviors or behavioral rules or constraints any way we want, even if they are, in our minds, orthogonal to the system's structure.

Incidentally, removing limitation (II), about separate whats and hows, also helps to deal with the classical question of completeness, that is, figuring out when we've finished the programming. The reason is that one way of describing play-in is that we can use it to program in the requirements directly, as part of the program. When we've finished doing that, we can be sure that nothing important has been left out, unless the requirements document left things out, something that in general no one can discover.

Play-out means employing powerful computing transparently to execute the grand total of all played-in pieces of behavior.

Also, regarding limitation (III), once we have liberated the programmer from having to divvy up the system's behavior along the lines of its structure, endless new possibilities open up. A central possibility involves changes and updates. For example, this style should make it possible to conveniently remove pieces of behavior that we don't like and replace them with others, which is quite different from replacing a tangible part of the system with some new one. I would love to be able to reprogram the interactions that the web-based systems I work with force me to follow—not to mention reprogramming my annoying and unnecessarily complicated DVD. I can't change the way Amazon or B&H respond to what I do, for example, but I can surely change everything that has to do with the way *my* browser and *my* computer deal with these websites. And how better to do that than by simply canceling some pieces of interactive behavior and playing in new ones, using the very interface on which we interact, subject, of course, to my inability to change their behavior?

As to inheritance, my take is that its most interesting (and eventually useful) facet involves substituting objects in ways that preserve specific interobject behaviors. A good example involves replacing your secretary. You don't mind working with another secretary with different ways of doing things (and indeed with a possibly different set of things he or she can do). However, you want to ensure that the new person will be allowed to replace the old one only if certain behaviors are preserved. Here you would have the flexibility of detaching the structure (the substitutability of an object) from behavior (the maintained/inherited behaviors).

If hard-pressed to say in a nutshell what the play-in idea embodies, I would emphasize the fact that it is intended to constitute a natural and smooth computerization of how we'd cause some entity to behave the way we want. The programmer teaches and guides the computer to get to "know" about the system's intended behavior under development. This is done by working with—nay, playing with—the system itself or some soft version of it, and it should be done in the way that most naturally reflects how the programmer thinks about that behavior. It can contain any number and any combination of complicated modality-rich pieces of behavior, which can in turn involve many pieces of the system, intermixed with local behaviors, and they can be temporally short or lengthy. These pieces should be allowed to express actual operational instructions, as well as examples, guidelines, rules and constraints, and so on. Whatever is natural for the programmer, and can be conveyed to the system under development by an intuitive hands-on process, should be allowed.

Of course, this sounds exceedingly naïve, offering little more than the simple statement that almost anything goes. And that's easy to say in a section about how to program, but it generates a heavy debt, one that will have to be repaid when we talk about how to *run* those "programs."

THE DREAM, PART 2: RUNNING THE PROGRAM

So now we have to discuss what happens during and after play-in. Although play-in doesn't *seem* to require coding in some language, the play-in process itself is a language of sorts, and it's something formal-looking that the computer understands will have to be generated as its result. Here "understand" must mean, at the very least, "knows how to execute/run."

How can we do that? Well, since we haven't yet left dream mode, I can still answer in lofty words: I believe we can, and should, harness all the power of computing to do exactly that.

To start with, play-in must be worked out in such a way that, as behavior is being played in, the process is somehow recorded. It would have to be subjected to the required on-the-fly processing, including possibly speech and natural-language recognition, and then to formalization and logical capture. This would have the effect of

transforming the play-in sequence into a formal artifact, whose semantics captures the properties of, or the constraints on, the allowed traces of system behavior.

In fact, we can view play-in as a means for coaxing temporal specifications of behavior out of the user, and a natural medium in which to formalize these specifications would be some version of *temporal logic*. This would then be the "code" resulting from the piece of programming carried out. Learning theory and other AI techniques might very well be needed here to intelligently generalize examples into generic behaviors and to become smarter at understanding the more elaborate behavior that might be played in later. Thus, play-in will have to be formulated as a "language" amenable to this kind of analysis.

My firm conviction (and experience) is that even very liberal and informal notions of play-in will yield to such a formalization approach, as long as the programmer can get immediate feedback about how the playing in has been rendered. This must include making it possible for the programmer to observe both the generated formal version of the played behavior as well as its immediate effect on the played interface.

This kind of consistency and realizability checking, as well as the computing of resulting behavior, does not seem wildly impossible.

The main thing however, is this: At any point in the play-in programming process—and a special case of this is when the programming is over and we have the final system that must start operating—the programmer can ask to run the current version of the program. This really means that powerful and heavy computing would be employed transparently to execute the grand total of all the played-in pieces of behavior. We have termed this process *play-out*,[6] and it should be doable in interpreter-style mode (direct execution) or in compiler-style mode (synthesizing an executable). The programmer should be able to play with the play-out, so to speak, moving around among the possibilities and narrowing things down—all naturally and intuitively. Of course, if needed, the programmer should also be allowed to make changes in the formal rendition.

Again, this is easier said than done. What does it mean to execute, or play out, the grand total of the played-in behaviors? This is best explained by going back to our earlier metaphor.

Not only do I dream of *programming* computers the way we educate children, teach students, or make good citizens, but also of *running* those programs the way we expect that those people then go off and proceed to live their lives. Whenever we feel as if we have given the system enough instructions and guidelines, we set it free to start behaving. It can then do whatever it feels like, as long as it adheres to whatever was programmed into it during play-in. Whatever we told it that it *must* do, it will indeed do; whatever we said it is *not allowed* to do, it will never do; whatever we said it *might* do (for example, a nondeterministic or probabilistic choice among several possibilities), it will decide whether to do or not in the appropriate fashion, and so on.

In fact, if we ourselves choose to be fully and pedantically obedient (something that, interestingly, humans cannot really be expected to be but computers can …), that would be exactly how we would manage our lives. We all have our "books" of rules, containing all manner of instructions, regulations, guidelines, and laws relevant to our existence, the elements of which come in a variety of degrees of detail and explicitness. If we choose to adopt the good citizen stand, we will carry out the algorithm just described: We'll live any way we choose, as long as it is within the confines of those books.

Can we reliably transfer this procedure to the formal realm of computing? And if so, what kind of computational tools would doing so require?

We must somehow generate actual behaviors of the system that are consistent with the collection of played-in pieces of behavior; this is often called *realizability*. And recall that these pieces are multimodal and can contain constraints as well as operational instructions, and thus may limit, or even contradict, each other, and the entire approach will clearly be highly nondeterministic. (Is this the in silico version of free will?)

The word "consistent" here is crucial. I believe that we could develop powerful computational techniques and use them to verify the consistency of what was played in, to compute and lay out a plethora of traces of behavior that are consistent with all of that, and then to choose which of them to actually run/execute. These computations should be doable on the fly, so that the programmer can be warned that a behavior being played-in contradicts what has already been programmed and to provide immediate feedback. More elaborate versions of such computations that could yield more efficient execution instructions could be done offline, in a compile-like mode. This difference is not that important to the issue itself.

What is interesting is that this kind of consistency and realizability checking, as well as the computing of resulting behavior, does not seem wildly impossible. Consider the former: What we are really talking about is a kind of *verification* problem, except that, since here the hows and the whats are intermixed; verifying one against the other is really just checking the consistency or realizability of the compound "program." Similarly, computing legal behaviors of the system, if indeed it has any, is often called *synthesis*, or *temporal synthesis*.[10]

Both verification and the related notion of synthesis have been the subjects of extensive research efforts,

Some Evidence of Feasibility: Scenario-Based Programming

Having had the play-in bug in mind for a long time, and wanting to see whether a play-in approach to specifying reactivity was at all possible, it became clear that we needed a language that was intuitive enough for engineers and programmers to use, but which was not limited to specifying behavior per object. The whole idea of play-in calls for freedom in talking about the interaction between the system's parts.

The turning point came in the 1998 collaboration with Werner Damm, which resulted in the language of *live sequence charts* (LSCs).[1] This is a temporal visual formalism that extends classical message sequence charts (MSCs), or their UML variant, sequence diagrams, mainly by being *multimodal*, allowing existential and universal flavors both for the charts themselves and for the internal elements. (For the latter these flavors are called *hot* and *cold*.) LSCs were defined in the natural framework of object-oriented systems, and are also expressible in temporal logic. They make it possible to talk in operational terms about the interaction between the system and its environment and among the system's objects. We use the term *interobject* for this and use the term *intraobject* for the more conventional object-by-object specifications. The language allows specifying scenarios of what can and might happen (like those of MSCs), but in the case of LSCs also what must happen, what is not allowed to happen, and much more.

For the next several years, an extensive collaboration started with then-PhD student Rami Marelly. That work addressed several issues.[2] The first was to strengthen the original version of LSCs considerably, increasing its expressive power by adding several crucial features. The main ones were the notion of time (and a sort of real time), and a notion of genericity via variables and symbolic instances. Genericity allows using the play-in process to specify by-example scenarios, such as making a specific phone call using specific objects (the numeric keys) but where the result of playing it in will be a generic chart that refers to any such objects and therefore captures making any call.

The two major results of the work with Marelly were *play-in* and *play-out* for the full language of LSCs, and the construction of the Play-Engine tool that supports the two techniques.[2] For play-in, we use a GUI for the system's objects (whose internal local methods have to be given up front, separately), and the kinds of play-in processes allowed are in line with the structure and flow of an LSC. So we essentially play-in an LSC via the GUI by clicking for activation, right-clicking for method and action menu selection, and so on. We use icons on the tools' interface to select hot or cold, symbolic or not, and other possibilities for the semantics of what is being played in. As we play in, the system generates and displays the corresponding LSC on the fly, and its effect on the GUI is shown continuously. Using the Play-Engine, we can actually play in the behavior of a mobile phone just as the text of this article describes, but the porcess is more rigid.

As to play-out, the child/student/citizen algorithm is implemented as is. The Play-Engine keeps track of all live (= active), or potentially live, LSCs simultaneously, including multiple live copies of the same chart with different object or value instantiations. At each step, the algorithm figures out what actions are possible as next steps, taking into account the entire set of possibilities, rules and constraints, from all the charts. Hot things are always done, cold ones might be done, and forbidden ones are never done. When a contradiction occurs—for example, a clash between something that has to be done and something that must not—the system reports a violation and stops. As in play-in, play-out is carried out on the GUI, which responds and provides full visual feedback about the run, and the executing LSCs are also animated in the background.

There is something very declarative and nondeterministic about LSCs. The basic "naïve" play-out mechanism deals with the nondeterminism inherent in the language just as most software development tools that execute models deal with racing conditions: It simply chooses one of the possible next things to do and does it. Of course, this can lead to violations, which could have been avoided had another path been taken instead. It could also have been avoided had we been able to carry out full temporal synthesis, since if the specification is known to be consistent—that is, realizable—there is a guaranteed way to make progress, and play-out need never fail.

Since synthesis is still a rather futuristic possibility, we came up with *smart play-out*.[2] In this technique, which is the heart of former student Hillel Kugler's PhD thesis, the tool translates the problem of finding a nonviolating superstep—that is, a sequence of actions that the

both on their limitations—for example, the undecidability of certain general versions and identifying decidable and tractable subcases—and on their efficiency and practicality in actual usage.[11,12] In addition to these, it also seems clear that planning algorithms, theorem proving, inductive logical reasoning, heuristics, and probabilistic techniques will turn out to be crucial.

Another major issue that needs to be raised is distribution and final code. It is all very well to say that we will

system takes in response to an external event—into a verification problem and then employs model-checking to solve it. The system then promptly executes the resulting superstep in a way that is transparent to the user. This is quite possibly the first use of hard-core verification not to prove properties of programs but to run those programs.

While smart play-out is currently limited to a single superstep, it is not too difficult to see that (in principle) extending the idea to unlimited depth of the tree of supersteps would be tantamount to solving the consistency/realizability problem for LSCs and could lead to play-out strategies that fail only if there is no other possibility. The lesson to be learned from smart play-out, I think, is this: We know that verification techniques are becoming very good at proving that programs do what we want; let's now harness them to help get those programs to do what we want.

Following this basic work, for which we use the term *scenario-based programming*, several more advanced pieces of work were carried out. We have recently devised another algorithm for executing LSCs, *planned play-out*, which uses AI-style planning algorithms to do essentially the same as smart play-out. The usage of planning here is not surprising, as planning can be viewed as a special case of synthesis. One benefit of planned play-out is that we can use it to find more than one possible superstep. Also, taking advantage of the fact that planned play-out works in interpreter mode, we have also implemented an exploration mechanism that allows the user to navigate among possibilities during execution, trying things out, backtracking, and so on.

Both these non-naïve methods—smart and planned play-out—follow the original play-out mechanism in that they are interpreter-style approaches to execution. However, we have not yet applied either of them to the full LSC language, with time and symbolic instances being the main features that cause difficulties. In contrast to this, we have also exploited the similarities between aspect-oriented programming and the interobject nature of LSCs to build a compiler for a variant of LSCs, which translates them into AspectJ. This is more than the usual kind of compiler-style downward translation: The LSCs are compiled into Java, to which are added what we call *scenario aspects* to coordinate the simultaneous monitoring and direct execution of the compiled LSCs. We can then compile the generated Java code and link it to a separately implemented Java program to create a single executable application.

In other work, we have investigated the possibility of synthesizing state machines for the separate objects from the LSCs; we have proposed a way to incorporate a behavioral and object hierarchy into the language to help scenario-based programming scale up. We also have worked out the corresponding enriched play-out technique, and we have devised a distributed play-out protocol for a subset of the LSC language. On a somewhat different note, we have built a linking tool, called *InterPlay*, which programmers can use to mix interobject behavior given in LSCs with separate behavior given for some of the objects in an intraobject language, such as conventional code or statecharts. We have also been investigating how LSCs and the Play-Engine fare in some specific application areas, such as telecommunication systems, tactical simulators, biological modeling, and web services.

Nevertheless, while definitely relevant to the dream of liberating programming from its three restraining straightjackets, all this work is still partial and preliminary. There are many serious issues that need to be resolved even if we restrict ourselves to the scenario-based language of LSCs and the relatively modest versions of play-in and play-out that have already been worked out. We have not yet dealt with consistency, or realizability, except in the limited scope of a single superstep, nor have we paid much attention to the optimization of the various execution mechanisms. And determining how to scale scenario-based programming up to large, multilevel systems will require more than an adequate definition of hierarchical LSCs and distributed play-out.

References

1. 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.
2. D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.

use all kinds of techniques to compute the "good citizen" live-by-the-rules idea of play-out, but there is a nagging feeling that a naïve approach to this—even if computationally very powerful—would have to somehow generate an overall controller or scheduler for the entire system. For real-world systems (again, we need only consider web applications as an alarming example), by the end of the day we are often required to have actual code running on separate machines, and possibly in separate locations, working together to constitute the running system.

It's nice to dream of specifying and executing behavior in a way that is orthogonal to the system's breakup into parts, but the final system implementation very often must adhere to the boundaries of those parts, and quite possibly this would include at least parts of the system's behavior too. So we will probably have to find ways to *distribute* the intuitive played-in behavior, breaking it up into pieces that the system's various parts can deal with. This kind of distributed play-out is highly nontrivial and calls for a distributed variant of the synthesis problem, which happens to be even harder and is not as well understood.[12] It will probably require the development of ever more powerful techniques and ideas connecting distributed and parallel computing with logic and verification-like methods.

ON SCENARIO-BASED PROGRAMMING

The material in the "Some Evidence of Feasibility: Scenario-Based Programming" sidebar is not imaginative—it is real work that has been done over the past nine years jointly with a group of greatly talented colleagues and students. When compared to the grandiose spirit of the previous comments, however, it is partial, fragmented, and rather narrow. For example, it is restricted to programming the reactive and interactive dynamics of sets of objects, and it doesn't attempt to deal with any algorithmic or data-intensive types of programming. Thus, even the potential scope of the dream discussed here does not become clear from it. Nevertheless, I maintain that it provides some evidence that it might be worth thinking more seriously about liberating programming along the lines I have discussed here.

In particular, using the language of LSCs and the Play-Engine discussed briefly in the sidebar, we can actually play in behavior similar to the cell phone example described earlier, albeit far more rigidly, and we can then play out the set of behaviors according to the "good citizen" algorithm. And all this is done in an interobject, rather than intraobject, fashion. In some of our more recent work, we have used well-known techniques from verification and AI, and this also adds to the feeling of feasibility that it raises.[8]

The bottom line is this: I believe there is no reason why we shouldn't make great efforts to bring widely researched and deeply worked-out ideas in computer science to bear upon the most basic and profound activity that involves computers, namely, programming them and running the resulting programs. Once liberated, programmers will probably have new kinds of work to do, possibly including the need to set up specialized features of the new sophisticated computational tools that would be running in the background.

There is obviously a great deal more to programming than specifying the reactive and interactive give and take of objects. I can imagine some ways in which the ideas described (or hallucinated about) here might be extended to other kinds of programming, involving other kinds of entities, such as classical algorithmics, data structures, and databases. But I'd be the first to admit that there are many more things that are relevant to all of this, about which I don't even know enough to dream of, let alone to imagine how to do. Also, I am not saying that any of this is easy, or even that it is clear that it can be done. Such is the nature of dreams.

On the other hand, dreaming and sharing the dreams with others has never been a mortal sin. ■

Acknowledgments

I would like to express my deepest thanks to the many dedicated and talented people whom I have had the pleasure of working with on scenario-based programming over the past few years. Special thanks go to Werner Damm, Rami Marelly, and Hillel Kugler, without whose lengthy collaboration even the dreaming would have been impossible. The other members and ex-members of my group who were actively involved in developing the ideas described in the sidebar include Shahar Maoz, Yoram Atir, Dan Barak, Asaf Kleinbort, Ron Merom, Ouri Poupko, and Itai Segall. Thanks also to Shahar Maoz and Moshe Vardi for comments on the manuscript. This article was written during a visit to the School of Informatics at the University of Edinburgh and was supported by a grant from the EPSRC. The research described in the sidebar was supported in part by the John von Neumann Center for the Development of Reactive Systems at the Weizmann Institute of Science, by a Minerva grant, by a collaborative NIH grant, and by the Israel Science Foundation.

References

1. D. Harel, "From Play-In Scenarios to Code: An Achievable Dream," *Computer*, Jan. 2001, pp. 53-60. Also published in *Proc. Fundamental Approaches to Software Engineering* (FASE 00), LNCS 1783, Springer-Verlag, 2000, pp. 22-34.
2. J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Comm. ACM*, vol. 21, no. 8, 1978, pp. 613-641.
3. M. Zloof, "Query-by-Example: A Data Base Language," *IBM Systems J.*, vol. 16, 1977, pp. 324-343.
4. K. Apt, *From Logic Programming to PROLOG*, Prentice Hall, 1996.
5. C. Simonyi, *The Death of Computer Languages, The Birth of Intentional Programming*, tech. report MSR-TR-95-52, Microsoft Research, 1995.
6. K. Marriott and P.J. Stuckey, *Programming with Constraints: An Introduction*, MIT Press, 1998.

7. R.E. Filman et al., *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
8. D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.
9. D. Harel and A. Pnueli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems*, K.R. Apt, ed., NATO ASI Series, vol. F-13, Springer-Verlag, 1985, pp. 477-498.
10. A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," *Proc. 16th ACM Symp. Principles of Programming Languages*, ACM Press, 1989, pp. 179-190.
11. E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 2000.
12. M.Y. Vardi, "From Verification to Synthesis," presentation, 2006 (in PostScript); www.cs.rice.edu/~vardi/papers/fmco06.ps.gz.

David Harel *is the William Sussman Professor in the Department of Computer Science and Applied Mathematics at the Weizmann Institute of Science. Harel invented statecharts, coinvented live-sequence charts, and was a member of the team that designed Statemate and Rhapsody. He also codeveloped the play-in/play-out approach to scenario-based programming and the Play-Engine. Recently, he has also been working on odor communication and biological modeling. He is a fellow of the IEEE, the AAAS, and the ACM. Contact him at dharel@weizmann.ac.il.*