From Play-In Scenarios to Code: An Achievable Dream

A development scheme for complex reactive systems leads from a user-friendly requirements capture method, called play-in scenarios, to full behavioral descriptions of system parts, and from there to final implementation.

David Harel
The Weizmann Institute of Science

In a 1992 *Computer* article,[1] I tried to present an optimistic view of the future of development methods for complex systems. Research since then only supports this optimism, as I will attempt to show.

This article presents a general, rather sweeping development scheme, combining ideas that have been known for a long time with more recent ones. The scheme makes it possible to go from a high-level user-friendly requirements capture method—which I call play-in scenarios—via a rich language for describing message sequencing to a full model of the system, and from there to final implementation.

A cyclic process of verifying the system against requirements and synthesizing system parts from the requirements is central to the proposal. The article puts special emphasis on the languages, methods, and computerized tools that allow smooth but rigorous transitions between the various stages of the scheme. In contrast to database systems, this article focuses on systems that have a dominant reactive, event-driven facet. For these systems, modeling and analyzing behavior is the most crucial and problematic issue.

MODELING THE SYSTEM

Over the years, the main approaches to high-level system modeling have been structured-analysis/structured-design (SA/SD) and object-oriented analysis and design (OOAD). The two modeling approaches are about a decade apart in initial conception and evolution. Over the years, both approaches have yielded visual formalisms for capturing the various parts of a system model, most notably its structure and behavior. The linking of structure and behavior is crucial and by no means a straightforward issue. In SA/SD, for example, each system function or activity is associated with a state machine or a statechart[2] that describes its behavior. In OOAD, as evident in the Unified Modeling Language (UML)[3] and its executable basis, the XUML,[4] each class is associated with a statechart, which describes the behavior of every instance object. The "Structured Analysis and Structured Design" and "Object-Oriented Analysis and Design" sidebars give some background on these modeling approaches.

An indispensable part of any serious modeling approach is a rigorous semantical basis for the model constructed—notably, for the behavioral parts of the model and their connection with the structure. It is these semantics that lead to the possibility of executing models and running actual code generated from them. (The code need not result in software; it could be in a hardware description language, leading to real hardware.) Figure 1 shows system modeling with full code generation.

Obviously, if we have the ability to generate full code, we would eventually want that code to serve as the basis for the final implementation. Some current tools, like Statemate and Rhapsody from I-Logix Inc. or Rose RealTime from Rational Corp., can in fact produce quality code, good enough for the implementation of many kinds of reactive systems. And there is no doubt that the techniques for this kind of "supercompilation" from high-level visual formalisms will improve in time. Providing higher levels of abstraction with automated downward transformations has always been the way to go, as long as the engineers who do the actual work are happy with the abstractions.

An early version of this article appeared in *Proc. Fundamental Approaches to Software Eng. (FASE)*, Lecture Notes in Computer Science, vol. 1783, Springer-Verlag, Berlin, Mar. 2000, pp. 22-34.

0018-9162/01/$10.00 © 2001 IEEE

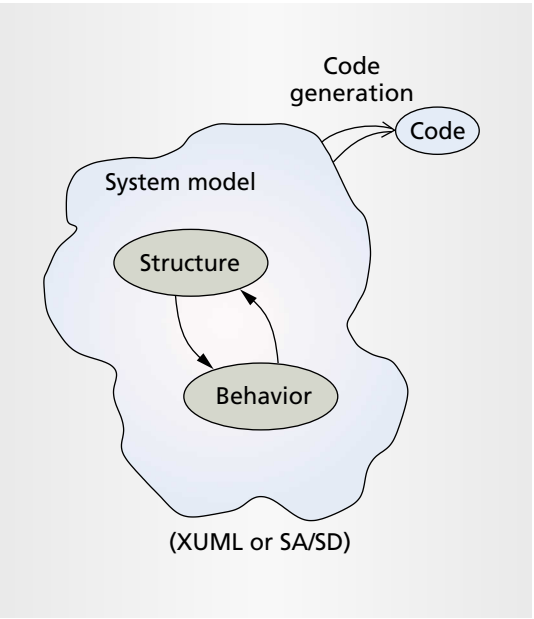

Figure 1. System modeling with full code generation. The system model consists of structure and behavior depicted using visual formalisms in executable UML (XUML) or structured analysis/structured design (SA/SD). A rigorous semantical basis makes it possible to automatically generate full runable code from the model.

SPECIFYING REQUIREMENTS

When developing a complex system, it is very important to be able to test and debug the model before investing extensively in implementation—hence, the desire for executable models.[1]

Requirements are the basis for testing and debugging by executing the model. By their very nature, requirements constitute the constraints, desires, and hopes we entertain concerning the system under development. We want to make sure, both during development and when we feel development is over, that the system does, or will do, what we intend or hope for it to do.

Figure 2 shows system modeling with full code generation and soft links to requirements. Requirements can be formal (rigorously and precisely defined) or

Structured Analysis and Structured Design

SA/SD, which started in the late 1970s, is based on raising classical procedural programming concepts to the modeling level and using diagrams, or visual formalisms, as the languages for modeling system structure. Structural models are based on functional decomposition and information flow and are depicted by hierarchical dataflow diagrams.

Many methodologists were instrumental in setting the ground for the SA/SD paradigm by devising the functional decomposition and dataflow diagram framework, including Tom DeMarco[1] and Larry Constantine and Ed Yourdon.[2] David Parnas's work over the years was very influential, too.

In the mid-1980s, three methodology teams—Ward and Mellor,[3] Hatley and Pirbhai,[4] and the Statemate team[5]—enriched this basic SA/SD model by using state diagrams or the richer language of statecharts[6] to add behavior to these efforts. A state diagram or statechart is associated with each function or activity, describing its behavior. Many nontrivial issues had to be worked out to properly connect structure with behavior, enabling modelers to construct a comprehensive and semantically rigorous model of the system. It is not enough to simply decide on a behavioral language and then associate each function or activity with a behavioral description. (This would be like saying that when you build a car, all you need are the structural things—body, chassis, wheels, and so on—and an engine, then you merely stick the engine under the hood and you are done.) The three teams struggled with this issue, and their decisions on how to link structure with behavior ended up being very similar. Careful behavioral modeling and its close linking with system structure are especially crucial for reactive systems,[7,8] of which real-time systems are a special case.

Statemate, released in 1987, was the first commercial tool to enable model execution and code generation from high-level models[5] (http://www.ilogix.com). *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*[9] gives an updated and detailed summary of the SA/SD languages, their relationships, and the way they are embedded in Statemate.

Of course, modelers don't need to adopt state machines or statecharts to describe behavior. Many other possible languages can be linked with the SA/SD structural diagrams. They include visual formalisms such as Petri nets or SDL diagrams and more algebraic methods like Communicating Sequential Processes or Calculus of Communicating Systems.

References

1. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.
2. L.L. Constantine and E. Yourdon, *Structured Design*, Prentice Hall, Upper Saddle River, N.J., 1979.
3. P. Ward and S. Mellor, *Structured Development for Real-Time Systems*: *Volumes 1-3*, Yourdon Press, New York, 1985.
4. D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1987.
5. D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Soft. Eng.*, vol. 16, no. 4, 1990, pp. 403-414; also in *Proc. 10th Int'l Conf. Soft. Eng.*, IEEE Press, Piscataway, N.J., 1988, pp. 396-406.
6. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, 1987, pp. 231-274; also tech. report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, 1984.
7. D. Harel and A. Pnueli, "On the Development of Reactive Systems," in *Logics and Models of Concurrent Systems*, K.R. Apt, ed., NATO ASI Series, vol. F-13, Springer-Verlag, New York, 1985, pp. 477-498.
8. A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," *Current Trends in Concurrency,* J. de Bakker et al., eds., Lecture Notes in Computer Science, vol. 224, Springer-Verlag, Berlin, 1986, pp. 510-584.
9. D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, New York, 1998.

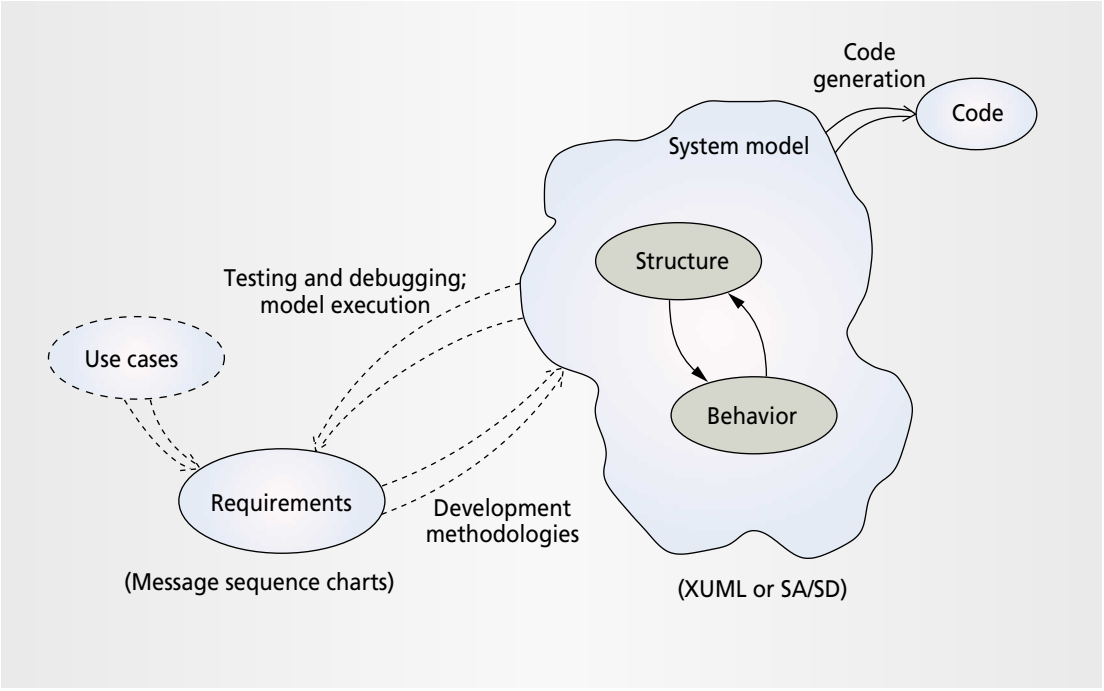

Figure 2. System modeling with "soft" links to requirements. Developers check the model against requirements by executing the model (testing and debugging). Developers use various methodologies to go from the requirements to the model. In general, however, these processes are not comprehensive, not guaranteed to succeed, and not fully automated.

Object-Oriented Analysis and Design

The late 1980s saw the first proposals for object-oriented analysis and design (OOAD). As in SA/SD, the basic idea in modeling system structure was to lift concepts up from the programming to the modeling level and to use visual formalisms. Inspired by entity-relationship diagrams,[1] several methodology teams recommended various forms of class and object diagrams for modeling system structure.[2-5] To model behavior, most object-oriented modeling approaches adopted statecharts.[6] Each class has an associated statechart, which describes the behavior of any instance object.

The issue of connecting structure and behavior is subtler and more complicated in the OOAD world than in the SA/SD world. Classes represent dynamically changing collections of concrete objects. Behavioral modeling must thus address issues related to object creation and destruction, message delegation, relationship modification and maintenance, aggregation, inheritance, and so on.

The links between behavior and structure must be defined in sufficient detail and with enough rigor to support the construction of tools that enable model execution and full code generation. Only a few tools have been able to do this. One is ObjectTime, which is based on the Real-Time Object-Oriented Modeling method[5] and is now part of the Rational RealTime tool (see http://www.rational.com).

Another tool is Rhapsody (http://www.ilogix.com), which is based on the work of Eran Gery and David Harel on executable object modeling with statecharts.[7] This work centers on a carefully constructed language set that includes class/object diagrams adapted from the Booch method[2] and the OMT method,[4] driven by statecharts for behavior.

This pair of languages also serves as the executable heart of the Unified Modeling Language,[8] put together by a team led by Grady Booch, James Rumbaugh, and Ivar Jacobson, which the Object Management Group adopted as a standard in 1997 (see http://www.omg.org). The class/object diagrams and the statecharts part of the UML is often called XUML (for executable UML). Thus, XUML is the part of UML that specifies unambiguous, executable, and therefore implementable, models.

UML has several means for specifying more elaborate aspects of system structure and architecture (for example, packages and components). An important part of UML for specifying requirements is Jacobson's use cases.[9]

References

1. P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Trans. Database Systems*, vol. 1, no. 1, 1976, pp. 9-36.
2. G. Booch, *Object-Oriented Analysis and Design, with Applications*, 2nd ed., Benjamin-Cummings, San Mateo, Calif., 1994.
3. S. Cook and J. Daniels, *Designing Object Systems: Object-Oriented Modeling with Syntropy*, Prentice Hall, Upper Saddle River, N.J., 1994.
4. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, Upper Saddle River, N.J., 1991.
5. B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.
6. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, 1987, pp. 231-274; also tech. report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, 1984.
7. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Computer*, July 1997, pp. 31-42.
8. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, Reading, Mass., 1999.
9. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley, Reading, Mass., 1992.

A collection of MSC scenarios cannot be considered an implementable system model. How would such a system operate? What would it do under general dynamic circumstances?

informal (written, say, in natural language or pseudocode). However, because this article is concerned mainly with processes that can be automated, the focus is on formal requirements.

Ever since the early days of high-level programming, computer science researchers have grappled with the question of how to best state what we want of a complex program or system. Notable efforts include the classical Floyd/Hoare invariant assertions method, with its pre- and postconditions and termination statements[5] and the many variants of temporal logic.[6]

These efforts make it possible to express the two main kinds of requirements of interest in modeling reactive systems. The first requirement is *safety*, which says that a bad thing can't happen; for example, this program will never terminate with the wrong answer, or this elevator door will never open between floors. The second requirement is *liveness*, which says that good things must happen. For example, this program will eventually terminate, or this elevator will open its door on the desired floor within the allotted time limit.

Scenarios and use cases

A more recent way to specify requirements, which is popular in the realm of object-oriented systems, is to use message sequence charts (MSCs). The International Telecommunication Union (the ITU, formerly the CCITT) adopted this visual language as a standard long ago.[7] The idea of MSCs also manifests itself in UML, but in a slightly weaker way, as the language of sequence diagrams.[3]

Both MSCs and UML's sequence diagrams specify scenarios as sequences of message interactions between object instances. This approach meshes very nicely with use cases,[8] the informal statement of the possible ways the system can be used: In the early stages of system development, engineers typically come up with use cases and then specify the scenarios that instantiate them. Scenarios capture the desired relationships among the processes, tasks, or object instances—and among these factors and the environment—in a way that is linear or quasilinear in time. (I include tasks and processes because, although much of this discussion is couched in the terminology of object-orientation and UML, there is nothing specific to objects in my arguments.) In other words, the modeler uses MSCs to specify the scenarios, or "stories," that the final system should—and hopefully will—satisfy and support, and these scenarios are instantiations of the more abstract and generic use cases.

REQUIREMENTS VERSUS THE SYSTEM MODEL

As Figure 2 shows, use cases and sequence charts are not part of the system, but rather part of the requirements from the system. They are constructed to capture the scenarios we would like the system to satisfy when implemented.

It is interesting to compare the interobject "one-story-for-all-objects" approach that sequence charts reflect with the dual intraobject "all-stories-for-one-object" approach manifest in the XUML modeling of objects using statecharts. In contrast to scenarios, modeling with statecharts is typically carried out at a later stage, and results in a full behavioral specification for each object instance (or task or process), providing details of its behavior under all possible conditions and in all possible stories provided in the interobject sequence charts. Because it is directly implementable, the intraobject specification is at the heart of the system model in Figures 1 and 2; ultimately, the final software will consist of code specified for each object.

In contrast, a collection of MSC scenarios cannot be considered an implementable system model. How would such a system operate? What would it do under general dynamic circumstances? Thus, MSCs and UML's sequence diagrams provide the behavior requirements, stating how the system should behave when implemented; statecharts—as linked to the class diagrams in XUML—provide the implementable behavior itself.

By and large, the literature does not clearly define the subtle difference between the two. Again and again, I come across articles and books that use the same phrases to introduce sequence diagrams and statecharts. At one point, such a publication might say, "sequence diagrams can be used to specify behavior," and later, "statecharts can be used to specify behavior." The reader is told nothing about the fundamental difference between the two—that sequence diagrams are a medium for conveying requirements and statecharts are part of the system model—or about the very different ways they are to be used. Naive readers often are confused and puzzled by the multitude of diagram types in the full UML standard and the lack of clear recommendations about what specifying a system means.

Moving between the two

In Figure 2, the arrows between "requirements" (lower left) and "system model" are dashed because they do not represent rigorous, comprehensive, computer-supported processes. Going from the requirements to the model is a long-studied issue, and many system development methodologies provide guidelines, heuristics, and sometimes carefully worked out, step-by-step processes for this. However, as good and useful as these processes are, they are soft methodological recommendations for how to proceed, not rigorous and automated methods.

The arrow going from the system model to the requirements depicts testing and debugging the model against the requirements, using model execution. Here

is a nice way to do this using the Rhapsody tool: Assume the user has specified the requirements as a set of sequence diagrams, perhaps instantiating previously prepared use cases. For simplicity, say that the result is diagram *A*. Later, when the system model has been specified in XUML, the user can ask Rhapsody to execute it. During execution, Rhapsody automatically constructs animated sequence diagrams, on the fly, showing the dynamics of object interaction as they actually happen during execution. Assume that this results in diagram *B*.

When this execution is completed, we can ask Rhapsody to compare diagrams *A* and *B* and to highlight any inconsistencies, such as contradictions in the partial order of events, or other differences, such as events appearing in one diagram but not in the other. In this way, Rhapsody helps debug the system's behavior against the requirements.

While this is a powerful and very useful way to check a system model's behavior, it is limited to executions we actually carry out, and thus suffers from the same drawbacks as classical testing and debugging. Because a system can have an infinite number of runs, some will always go unchecked, and it could be those unchecked runs that violate the requirements (in this case, by being inconsistent with diagram *A*). As Edsger Dijkstra put it years ago, "Testing and debugging cannot be used to demonstrate the absence of errors, only their presence." This softness of the debugging process is the reason the arrow from the system model to the requirements in Figure 2 is also dashed.

SEQUENCE CHARTS AND LIVE SEQUENCE CHARTS

As a requirements language, all known versions of MSCs, including the ITU standard[7] and the sequence diagrams adopted in the UML,[2] are extremely weak in expressive power. Their semantics are little more than a set of simple constraints on the partial order of possible events in a system execution. Virtually nothing can be said in MSCs about what the system will actually do when it runs. These diagrams can state what might possibly occur, not what must occur. Thus, amazingly, if you want to be a purist, under most definitions of the semantics of MSCs, an empty system—one that doesn't do anything in response to anything—satisfies any such chart. So just sitting back and doing nothing will satisfy your requirements. (Usually, however, there is a minimal, often implicit, requirement that at least one run of the system should wind its way correctly through each one of the specified sequence charts.)

Another troublesome drawback of MSCs is their inability to specify unwanted scenarios. We want to forbid the occurrence of these antiscenarios, and they are crucial in setting up safety requirements.

A recent paper addressed these deficiencies and proposed an extension of MSCs, called live sequence charts (LSCs).[9] As the name implies, LSCs specify *liveness*, things that must occur. They let modelers distinguish between possible and necessary behavior both globally, on the level of an entire chart, and locally, when specifying events, conditions, and progress over time within a chart. The live, or hot, elements also make it possible to specify antiscenarios. The other elements, termed cold, support branching and iteration.

Showing how LSCs deal with conditions or guards gives a flavor of how they work: Assume that *P* is a hot condition appearing at a certain location in the chart. Then *P* must be true if and when that location is reached during a system run, and if it is not, the system aborts. In other words, *P* really must be true; otherwise there is an unforgivable error. In this way, modelers can specify antiscenarios (an elevator door opening when it shouldn't or a missile firing when the radar is not locked on the target). In contrast, if *P* is a cold condition, then it also should be true if and when the location is reached, but if it is not true, there is no catastrophe. Rather, the execution merely exits, and we simply move up one level—out of the present chart if *P* is on the top level of that chart, or out of the subchart and continuing from outside of it if *P* is inside a subchart block. This makes it possible to specify control structure constructs, such as if-then-else and while-do, using *P* as a controlling guard.

LSCs make it possible to start looking more seriously at the dichotomy of reactive behavior—the relationship between the interobject requirements view and the intraobject system model.

It is not yet clear whether LSCs are exactly what we need, and more work is definitely required. We need to gain experience using the language and we must build good implementations. Nonetheless, relative to MSCs, LSCs offer a far more powerful way to visually specify behavioral requirements. Because their expressive power is far greater (essentially that of XUML itself), LSCs also make it possible to start looking more seriously at the dichotomy of reactive behavior—the relationship between the interobject requirements view and the intraobject system model.

VERIFICATION AND SYNTHESIS

In Figure 3, the two dashed arrows between the requirements and the model have been made solid. This means we have at our disposal hard, formal, and rigorous—and mainly fully automatable—links between the system model (in XUML,[4] for example, or in a suitable version of SA/SD) and the requirements (in LSCs,[9] for example, or in temporal logic[6] or timing diagrams[10]).

From model to requirements

Going from the system model to requirements, instead of testing and debugging by executing models, we are interested in using true verification to check the

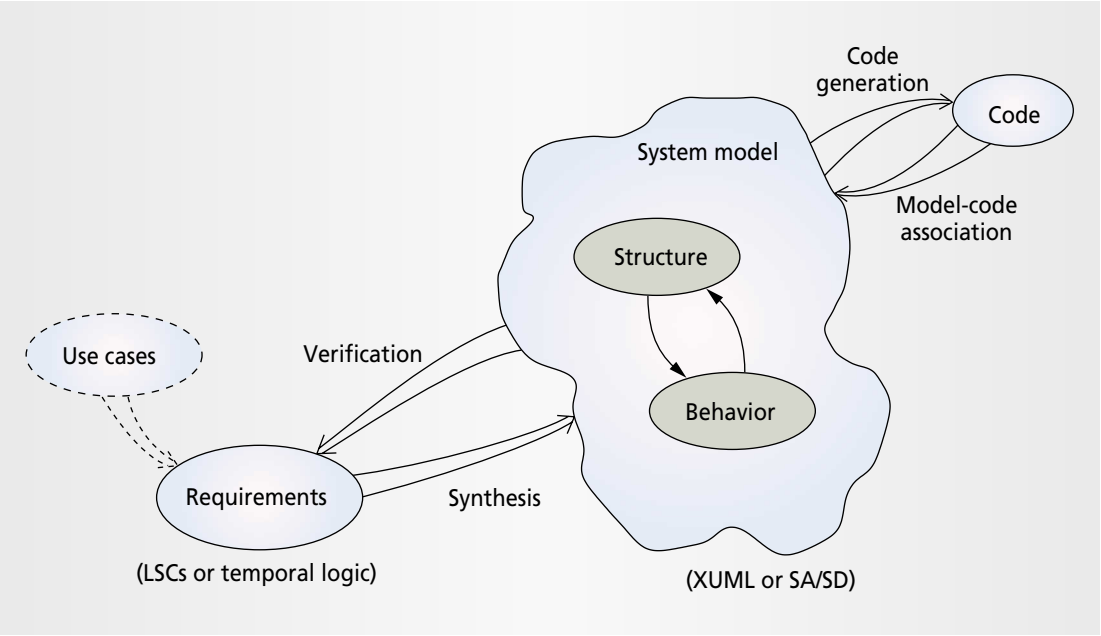

Figure 3. System modeling with "hard" links to requirements. This is a major part of the dream, in which true verification checks the model against requirements, and synthesis goes from the requirements to the model. These processes, when available, will be comprehensive, guaranteed to succeed, and fully automated.

system model against the requirements. This is not what CASE tool people in the 1980s often called "validation and verification," which did not amount to much more than consistency checking of the model's syntax. Rather, it is a mathematically rigorous and precise proof that the model satisfies the requirements, and a computerized verifier automatically does the proof.

Because we are using powerful languages like LSCs (or the analogous temporal logics or timing diagrams), this requires more than merely executing the system model and making sure that the sequence diagrams you get from the run are consistent with the ones you prepared in advance. It means making sure, for example, that the things an LSC says must not happen (the antiscenarios) will indeed never happen, and the things it says must happen (or must happen within certain time constraints) will indeed happen. These are facts that, in general, no amount of execution can verify.

Although general verification constitutes a noncomputable algorithmic problem, the idea of rigorously verifying programs and systems—hardware and software—has come a long way since the pioneering work on invariant assertions and the later work on temporal logic and model checking. These days we can safely say that we can carry out true verification in many cases, especially in the finite-state cases that arise in reactive real-time systems.

I-Logix has recently produced a version of the Statemate tool with verification capabilities that it will release as a product in the near future. Doing the same for an OOAD tool like Rhapsody is just a matter of time. Before long, I believe, we will be routinely using automated tools to verify models against requirements.

From requirements to model

In the opposite direction, going from the requirements to the model, we have synthesis. Instead of guiding system developers in informal ways to build models according to their desires and hopes, we would like our tools to be able to synthesize directly from those desires and hopes, if they are indeed implementable. We want to be able to automatically generate a system model from the requirements. (For the sake of the discussion, I assume that the structure—the division into objects or components, for example—has already been determined.)

This is much harder than synthesizing code from a system model, which is really only a high-level kind of compilation. The duality between the scenario style (requirements) and the statechart style (modeling) in saying what a system does over time renders the synthesis of an implementable system model from sequence-based requirements a truly formidable task. It is not too hard to do this for the weak MSCs, which can't say much about what we really want from the system. It is much more difficult for more realistic requirements languages, such as LSCs or temporal logic.

How then can we synthesize a good first approximation of the statecharts from the LSCs? Several researchers have addressed similar issues, resulting in work on certain kinds of synthesis from timing diagrams[10] and temporal logic.[11] A recent paper describes a first-cut attempt at algorithms for synthesizing state-machines and statecharts from LSCs (albeit, in a slightly restricted setup and for the time being yielding very large models).[12] The method first determines whether the requirements are consistent (whether any existing system model satisfies them) and then uses the fact that being consistent and having a model (being implementable) are equivalent notions to synthesize an actual model.

Much rather deep research into this issue is currently in progress. I believe that synthesis will eventually end up like verification—hard in principle but not beyond a practical and useful solution.

From code to model

Figure 3 also contains a solid arrow going from the code to the system model ("model-code association" in

the upper right). This indicates the developer's ability to make a round-trip back from the code to the model. Making certain kinds of changes in the code automatically reflects back as changes in the model's visual formalisms. This renders the classical cycle of activities that takes place between design and implementation easier and less error prone. Rhapsody provides a useful form of this model-code association. There is reason to believe that this ability will be commonplace in the future and that the applicability of the techniques enabling it will become broader and more powerful.

HOW SHOULD DEVELOPMENT PROCEED?

Figure 3 appears to imply that we wouldn't need the arrows going from right to left at all, and developers could do without verification, testing, or model-code association. A system developer could go directly and smoothly from desires to results: State your requirements, get your tool to synthesize the system model, get it to generate code from the model, and you are all set.

Obviously, this is not the case. We will always need to develop systems incrementally, with various cycles of activity taking place, possibly according to the spiral philosophy of development. Such a methodology calls for cycles of development, producing continuously refined and extended versions of the system. One cycle would be between the requirements and the model, incrementally extending and refining the system under development by following the dashed arrows of Figure 2—development methodologies, testing, and debugging—and the solid arrows of Figure 3—synthesis and verification. The other (less significant) cycle would be the same as the requirements-model cycle, but it would be between the model and the implementation in code, repeatedly fine-tuning the final artifact.

While we eventually might need to modify the classical life cycle approaches somewhat, I have not worked out a full step-by-step methodology for how to proceed in developing a system. Rather, this article describes the various parts of such a methodology, the languages and tools they involve, and their interrelationships. To propose a full-blown methodology, we will need more than a few examples appearing in hastily written methodology books that are wisdom-rich but technically shallow. This will not happen overnight. Rather, we will need to rely on the profound knowledge and rigor that will accumulate from years of experience in using these techniques with their full semantic underpinnings, with support from truly powerful tools.

PLAY-IN SCENARIOS

To complete the dream I have sketched, we need one last piece: a far more convenient way to set up behavioral requirements, suitable not only for system engineers but also for their clients, such as users and contractors.

Toward that end, I propose play-in scenarios. When you execute a model, you play *out* a scenario. This becomes apparent when you use the tool to execute models interactively. It becomes especially transparent and impressive (useful, too) when you work with a soft panel mock-up of the system's final interface or even with the system's actual hardware, as is possible in the tools mentioned earlier. You can play out a scenario by standing in, so to speak, for the system's environment, introducing events and changes in values and observing the results as they unfold.[1]

In contrast, here we would play *in* scenarios. This is done prior to building any behavioral model of the system in order to set up the requirements, perhaps driven by use cases. Rather than using conventional languages, visual or otherwise, to specify scenarios, modelers work directly opposite a mock-up of the system's interface, using a highly user-friendly method of teaching their tool about the desired and undesired scenarios. Developers can do this work together with clients or potential users, expediting the process and eliminating many kinds of behavior-related errors that come up during the development process.

Rather than using conventional languages to specify scenarios, modelers work directly opposite a mock-up of the system's interface, using a highly user-friendly method of teaching their tool about the desired and undesired scenarios.

Think of a graphical image of a cellular phone, for example, appearing on the developer's computer screen. There is nothing beneath it: No behavior has been specified for it yet. You now start entering scenarios by clicking and dragging, playing in inputs and the system's responses, indicating whether things are hot or cold, instantiated or generic, and more. The interactive process also includes a means of refining the system's structure as the work progresses by forming composite objects and their aggregates and setting up inheriting objects.

As the process of playing in the scenario-based requirements continues, the underlying tool—the play-in engine—will automatically and incrementally generate the formal LSCs (not merely MSCs) or the temporal logic formulas that capture the played-in scenarios accurately. Instead of using abstract engineer-oriented languages, we apply a friendly, intuitive, user-oriented automated process to construct rigorous and comprehensive requirements.

Here, too, much research remains to be done. While the idea of play-in scenarios has a nontrivial mathematical/algorithmic side, a large part of the effort needed relates to its human aspects. There must be a powerful, yet natural and easy-to-use way to interact with an essentially behavior-free "system shell" so that we can tell it what we want from it. A doctoral student and I have been developing the first version of a play-in environment at the Weizmann Institute and hope to publish the details soon.

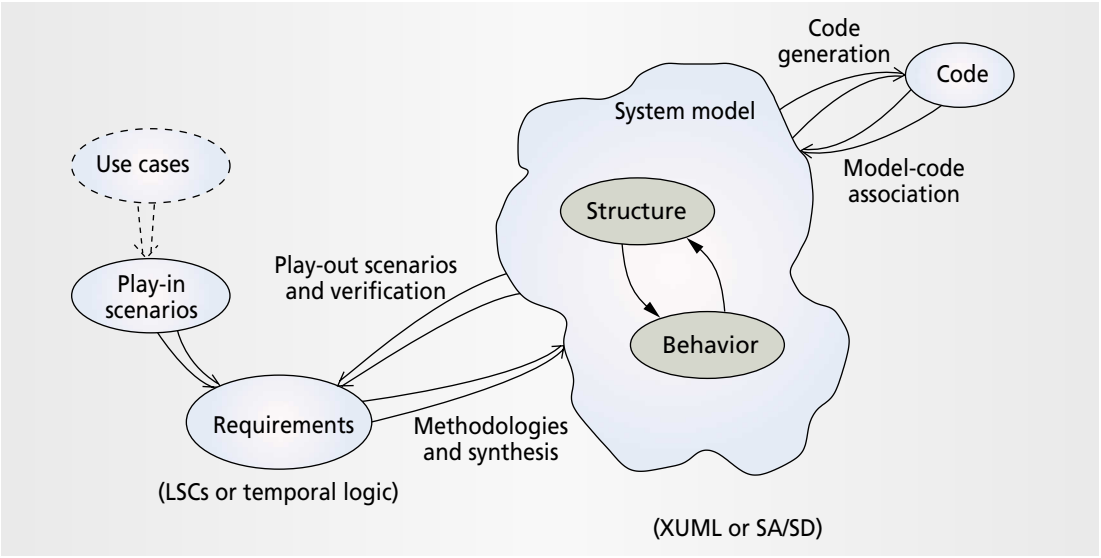

Figure 4. The dream in full. In addition to verification and synthesis, the dream calls for requirements to be "played in" directly opposite a mock-up of the system's interface, using a user-friendly method of teaching the system its behavior. Developers can play in scenarios together with clients or potential users.

Figure 4 summarizes the complete system development dream.

It is probably no great exaggeration to say that there is a lot more we don't know and can't achieve yet in this business than what we do know and can achieve. In addition to the topics I have presented, many issues require further research and development to be satisfactorily incorporated into the overall scheme. These include real-time analysis, automatic diagram layout, and dealing with hybrid systems that have both continuous and discrete facets.

The efforts of scores of researchers, methodologists, and language designers have resulted in more than we could have hoped for a decade or so ago, and for this we should be thankful and humble. There is still a long road ahead, but there is a dream in the offing. While several parts of this dream are not even close to being fully available, the dream is not unattainable. If it comes true, it could have a significant effect on the way we develop complex systems. ✷

References

1. D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development," *Computer,* Jan. 1992, pp. 8-20.
2. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, 1987, pp. 231-274; also tech. report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, 1984.
3. Unified Modeling Language (UML) documentation, Object Management Group (OMG), http://www.omg.org.
4. D. Harel and E. Gery, "Executable Object Modeling with Statecharts," *Computer*, July 1997, pp. 31-42.
5. A. Apt, *Verification of Sequential and Concurrent Programs,* 2nd ed., Springer-Verlag, New York, 1997.
6. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, 1992.
7. "MSCs: ITU-T Recommendation Z.120: Message Sequence Chart (MSC)," ITU-T, Geneva, 1996.
8. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley, Reading, Mass., 1992.
9. W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, submitted for publication. An early version was published in *Proc. 3rd IFIP Int'l Conf. Formal Methods for Open Object-Based Distributed Systems,* P. Ciancarini, A. Fantechi, and R. Gorrieri, eds., Kluwer Academic, New York, 1999, pp. 293-312.
10. R. Schlor and W. Damm, "Specification and Verification of System-Level Hardware Designs Using Timing Diagrams," *Proc. European Conf. Design Automation*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 518-524.
11. A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," *Proc. 16th ACM Symp. Principles of Programming Languages*, ACM Press, New York, 1989, pp. 179-190.
12. D. Harel and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications," *Proc. 5th Int'l Conf. Implementation and Application of Automata*, Lecture Notes in Computer Science, Springer-Verlag, New York, submitted for publication; also, tech. report MCS99-20, The Weizmann Institute of Science, Rehovot, Israel, Oct. 1999.

***David Harel** is the William Sussman Professor at The Weizmann Institute of Science in Israel and is dean of the faculty of mathematics and computer science. He is also cofounder and chief scientist of I-Logix and DigiScents Israel. His research interests are in computability and complexity theory, logics of programs, automata theory, visual languages, systems engineering, and, more recently, the mathematics and algorithmics of olfaction and smell communication. Harel received a PhD in computer science from the Massachusetts Institute of Technology. His most recent books are* Computers Ltd.: What They Really Can't Do *(Oxford University Press, London, 2000), and* Dynamic Logic *(with Dexter Kozen and Jurek Tiuryn, MIT Press, Cambridge, Mass., 2000). Harel is a Fellow of the ACM and the IEEE. Contact him at harel@wisdom.weizmann.ac.il.*