From Play-In Scenarios To Code: An Achievable Dream

David Harel*

IEEE Computer, to appear. Preliminary version in *Proc. Fundamental Approaches to Software Engineering* (*FASE*), Lecture Notes in Computer Science, Vol. 1783, Springer-Verlag, March 2000, pp. 22–34.

Abstract

A development scheme for complex reactive systems leads from a user-friendly method for playing in scenarios, to full behavioral descriptions of system parts, and from there to final implementation. The entire scheme, which includes a cyclic processe of verification and synthesis, should be supported by semantically rigorous automated tools.

Keywords: Code generation, Executable specifications, Play-in scenarios, Reactive system, Sequence charts, Software engineering, Statecharts, Synthesis, Verification.

In a 1992 *Computer* article [4], I tried to present an optimistic view of the future of development methods for complex systems. Research since then only supports this optimism, as the present article will attempt to show.

A general, rather sweeping development scheme is proposed, combining ideas that have been known for a long time with some more recent ones. The scheme makes it possible to go from a high-level, user-friendly requirements capture method, which we shall 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. We put a special emphasis on the languages, methods and computerized tools that allow for smooth but rigorous transitions between the various stages of the scheme.

In contrast to database systems, we shall be interested here in systems that have a dominant reactive, event-driven facet. For these, modeling and analyzing behavior is the most crucial and problematic issue. Such systems are often embedded systems with a real time facet.

*The Weizmann Institute of Science, Rehovot, Israel. Email: harel@wisdom.weizmann.ac.il.

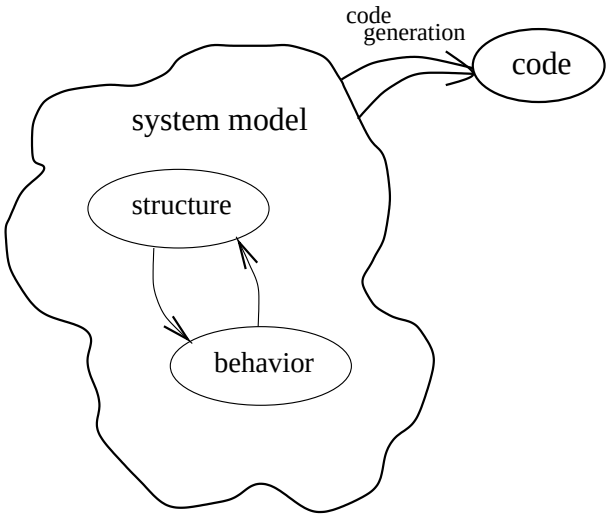

Figure 1: System modeling with full code generation

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 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 is 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 [3], which describes its behavior. In OOAD, as evident in the UML [12] and its executable basis, the XUML [5], each class is associated with a statechart, which describes the behavior of every instance object. See the boxes "Structured Analysis and Structured Design" and "Object-Oriented Analysis and Design".

An indispensible 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 this semantics that leads to the possibility of executing models and running actual code generated from them. See Figure 1. (The code need not necessarily result in software; it could be in a hardware description language, leading to real hardware.) 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., are in fact capable of producing 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 'super-compilation' from high-level visual formalisms will improve in time. Providing higher levels of abstraction with automated

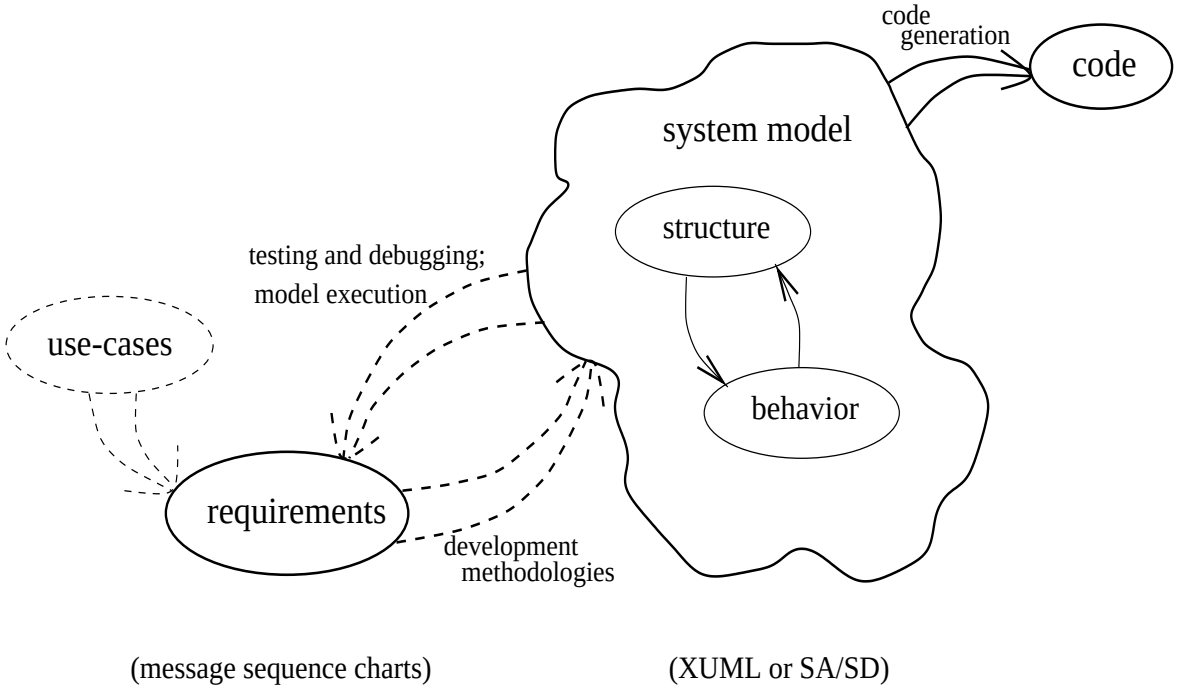

Figure 2: System modeling with 'soft' links to requirements

downward transformations has always been the way to go, as long as the abstractions are ones with which the engineers who do the actual work are happy.

Specifying requirements

When developing a complex system, it is very important to be able to test and debug the model prior to investing extensively in implementation. Hence the desire for executable models [4]. The basis for testing and debugging by executing the model are the requirements, which, by their very nature, 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. See Figure 2.

Requirements can be formal (rigorously and precisely defined) or informal (written, say, in natural language or pseudocode). However, since this article is concerned mainly with processes that can be automated, the focus will be 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 are those embodied in the classical Floyd/Hoare invariant assertions method, with its pre- and post-conditions and termination statements [1], and in the many variants of temporal logic [8]. These make it possible to express the two main kinds of requirements of

interest in complex modeling of reactive systems. The first is safety requirements, which say 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 type is liveness requirements, which say 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.

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 Telecommunications Union (the ITU, fomerly the CCITT) adopted this visual language as a standard long ago [9]. It also manifests itself in the UML in a slightly weaker way as the language of *sequence diagrams* (see [12]). MSCs, or UML's sequence diagrams, are used to specify scenarios as sequences of message interactions between object instances. This approach meshes very nicely with *use-cases* [7], 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. This captures the desired inter-relationships between the processes, tasks, or object instances — and between them and the environment — in a way that is linear or quasi-linear in time. (Tasks and processes are netioned here too, since although much of our discussion is couched in the terminology of object-orientation and the UML, there is nothing special to objects in the points being made.) In other words, the modeler uses MSCs to specify the scenarios, or the '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 vs. the system model

It is important to realize that use cases and scenarios are not part of the system. They are part of the requirements *from* the system, as Figure 2 shows. They are constructed in order to capture the scenarios that we would like our system will satisfy, when implemented.

It is interesting to compare the int*er*-object 'one-story-for-all-objects' approach that sequence charts reflect, with the dual int*ra*-object '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 inter-object sequence charts.

The intra-object specification is at the heart of the system model of Figures 1 and 2, since it is directly implementable; ultimately, the final software will consist of code specified for each object. In contrast, the requirements cannot be implemented. A collection of MSC scenarios cannot be considered an implementable model of the system: How would such a system operate? What would it do under general dynamic circumstances? Thus, MSCs, or UML's sequence diagrams, provide the requirements on behavior, stating our desires about how the system should behave when implemented, whereas statecharts — as linked to the class diagrams in the XUML — provide the implementable behavior itself.

Consider now the arrows in Figure 2 between the requirements and the system model. These arrows are dashed for a reason: 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 on 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, which is supported by the Rhapsody tool. Assume the user has specified the requirements as a set of sequence diagrams, perhaps instantiating previously prepared use-cases. For simplicity, let us say that this results in a diagram called A. Later, when the system model has been specified, the user can ask the Rhapsody tool to execute it in the following way. During execution, animated sequence diagrams will be automatically constructed, 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, Rhapsody can be asked 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 behavior of the system against the requirements.

While this is a powerful and very useful way to check the behavior of a system model, it is limited to those executions that we actually carry out, and thus suffers from the same drawbacks as classical testing and debugging. Since a system can have an infinite number of runs, some will always go unchecked, and it could be those that violate the requirements (in our case, by being inconsistent with diagram A). As Edsger Dijkstra famously 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 is dashed too.

Sequence charts and live sequence charts

Two points must now be made regarding sequence charts. The first is one of exposition: by and large, the subtle difference in the roles of sequence-based languages for behavior and component-based ones is not made clear in the literature. Again and again, one comes across articles and books in which the very same phrases are used to introduce sequence diagrams and statecharts. At one point such a publication might say that "sequence diagrams can be used to specify behavior", and later it might say that "statecharts can be used to specify behavior". Sadly, the reader is told nothing about the fundamental difference between the two — that one is a medium for conveying requirements and one is part of the system model — and about the very different ways they are to be used. This obscurity is one of the reasons many naive readers come away confused and puzzled by the multitude of diagram types in the full UML standard,

and the lack of clear recommendations about what it means to specify a system.

The second point is more substantial. As a requirements language, all known versions of MSCs, including the ITU standard [9] and the sequence diagrams adopted in the UML [12], are extremely weak in expressive power. Their semantics is little more than a set of simple constraints on the partial order of possible events in some possible system execution. Virtually nothing can be said in MSCs about what the system will actually do when run. These diagrams can state what might possibly occur, not what must occur. Thus, amazingly, if one wants to be puristic, then 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 make your requirements happy. (Usually, however, there is a minimal, often implicit, requirement that each one of the specificed sequence charts should have at least one run of the system that winds its way correctly through it.)

Another troublesome drawback of MSCs is their inability to specify unwanted scenarios. These *antiscenarios* are ones whose occurrence we want to forbid, and they are crucial in setting up safety requirements.

In a recent paper these deficiencies have been addressed, and an extension of MSCs, called *live sequence charts* (or *LSCs*) has been proposed [2]. As the name implies, LSCs can deal with specifying liveness, i.e., things that must occur. This is done by allowing the distinction 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, make it possible to also specify antiscenarios. The others, termed *cold*, can be used to support branching and iteration.

To give a flavor of how LSCs work, here is how they deal with conditions, or guards. 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 we have an unforgivable error. In this way, modelers can specify anti-scenarios (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 were 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 the 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.

It is not yet clear whether LSCs is exactly what we need, and a lot more work is definitely required. Experience using the language must be gained, and good implementations must be constructed. But the proposal for extending MSCs has now been made, rendering LSCs a candidate for a far more powerful way of visually specifying behavioral requirements. Since their expressive power is far greater than MSCs (it is essentially that of XUML itself), LSCs also make it possible to start looking more seriously at the aforementioned dichotomy of reactive behavior, namely, the relationship between the inter-object requirements view and the intra-

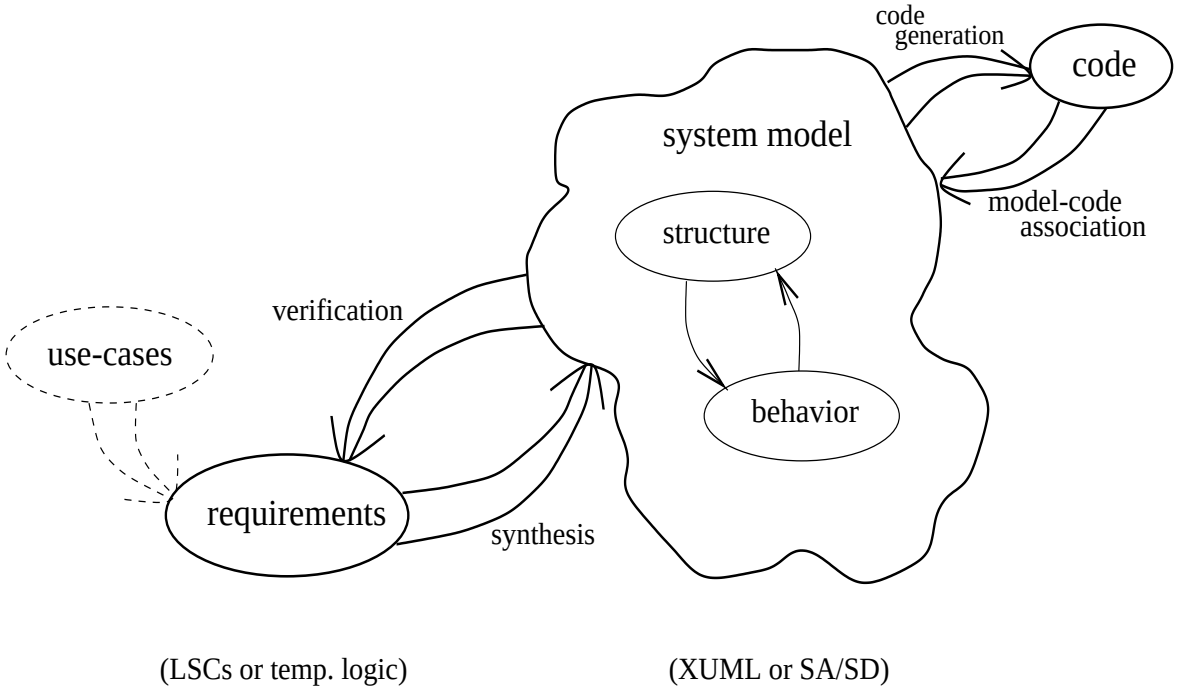

Figure 3: System modeling with 'hard' links to requirements

object implementable model view.

Verification and synthesis

Let us now consider Figure 3, in which the two dashed arrows between the requirements and the model have been made solid. We are now talking about the possibility of having at our disposal 'hard', formal and rigorous, and mainly fully automatable, links between the system model (in XUML [5], for example, or in a suitable version of SA/SD) and the requirements (in LSCs [2], for example, or in temporal logic [8] or timing diagrams [11]).

Going from the system model to requirements, instead of testing and debugging by executing models, we are interested in checking the system model against the requirements using true verification. 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. What we have in mind is a mathematically rigorous and precise proof that the model satisfies the requirements, and we want this to be done automatically by a computerized verifier. Since we are using powerful languages like LSCs (or the analogous temporal logics or timing diagrams) this means far, far 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 say must not happen (the anti-scenarios), 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.

This article is not a treatise on verification, so I will not say too much about it here. However, although verification in general constitutes a non-computable 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 true verification can be carried out in many, many cases, especially in the finite-state ones that arise in the realm of reactive real-time systems. A version of the Statemate tool with true verification capabilities has recently been constructed, and will be released as a product very shortly. Doing the same for an OOAD tool like Rhapsody or Rose RealTime is just a matter of time. Before long, I believe, we will be routinely using automated tools to verify models agains requirements.

In the opposite direction, going from the requirements to the model, we have synthesis. Instead of guiding system developers in informal ways to try to build models according to their dreams and hopes, we would very much like our tools to be able to synthesize directly from those dreams 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, we assume that the structure — the division into objects or components, for example — has already been determined.)

This is a whole lot harder than synthesizing code from a system model, which is really but 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 a lot more difficult for far more realistic requirements languages, such as LSCs or temporal logic.

How can we synthesize a good first approximation of the statecharts from the LSCs? Several researchers have addressed similar issues in the past, resulting in work on certain kinds of synthesis from temporal logic [10] and timing diagrams [11]. More recently, in [6], a first-cut attempt at algorithms for synthesizing state-machines and statecharts from LSCs has been reported (albeit, in a slightly restricted setup and for the time being yielding very large models). This is done by first determining whether the requirements are consistent (whether there exists any system model satisfying them), then proving that being consistent and having a model (being implementable) are equivalent notions, and then using the proof of consistency to synthesize an actual model. There is still a lot of rather deep research to be done here, and work is in progress as this is being written. I believe that synthesis will eventually end up like verification — hard in principle but not beyond a practical and useful solution.

Figure 3 also contains a solid arrow in the right-hand upper part of the figure, going from the code to the system model. This indicates the ability of the developer to 'round-trip' back from the code to the model. Making certain kinds of changes in the former reflects automatically

back as changes in the visual formalisms of the latter. Such an ability renders the classical cycle of activities that takes place between design and implementation easier and less error prone. A modest (but very useful) form of this model-code association is already available in the Rhapsody tool. There is reason to believe that this kind of ability will also be commonplace in the future, and that the techniques enabling it will become more powerful and far broader in applicability.

How should development proceed?

It is tempting to say that if the situation is as in Figure 3, we don't need the arrows going from right to left at all, and developers can do without verification or testing or model-code association. A system developer will be able to go directly and smoothly from desires to results: state your requirements, get your tool to synthesize the system model, then get it to generate code from the model, and you are all set!

Obviously, this is not the case. Systems will always have to be developed incrementally, with various cycles of activity taking place, possibly according to the spiral philosophy of development. Such a methodology would call for cycles of development, producing continuously refined and extended versions of the system. One kind of 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 and testing and debugging — and the solid ones of Figure 3 — synthesis and verification. The other (less significant) one would do the same between the model and the implementation in code, repeatedly fine tuning the final artifact.

While the more ambitious parts of this article implicitly suggest that the classical life cycle models might eventually have to be modified somewhat, I am not claiming to have worked out a full step-by-step methodology for how to proceed in developing a system, but to have discussed the various parts of such a methodology, the languages and tools they involve, and their inter-relationships. In order to be able 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. Rather, we shall have to rely on the deep and profound knowledge and rigor that will accumulate from years of experience in using these techniques. It will not happen overnight, even if all the required tools were just around the corner.

Play-in scenarios

To complete the dream I've tried to sketch, we need one last piece: a far more convenient way of setting up the behavioral requirements, suitable not only for system engineers but for their clients — such as the users and contractors. I would thus like to introduce an idea that I will call *play-in scenarios*. When you execute a model, you play-*out* a scenario. This becomes apparent when you use the tool's ability to execute models interactively, and it becomes

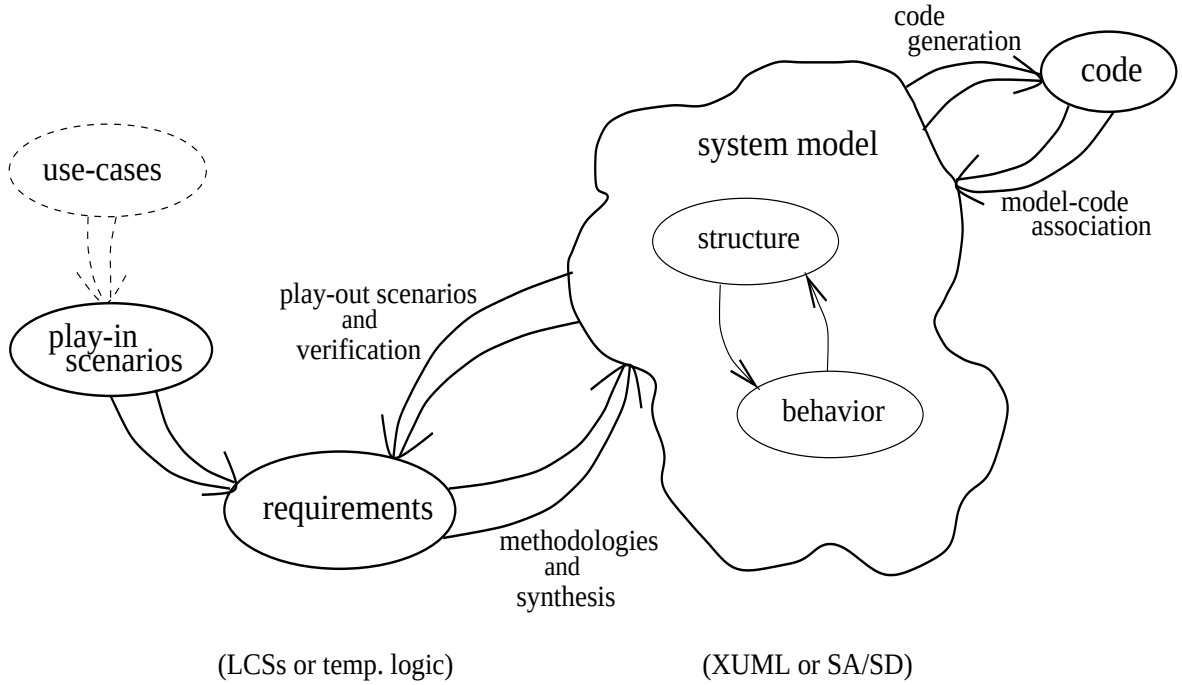

Figure 4: The dream in full

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 [4].

What I am proposing here is to play-*in* scenarios. This can be done prior to building any behavioral model of the system, in order to set up the requirements, perhaps driven by use-cases. The scenarios will not be specified using conventional languages, visual or otherwise, but by working directly opposite a mock-up of the system's interface, using a highly user-friendly method of 'teaching' your tool about the desired and undesired scenarios.

Think of a graphical image of cellular phone, for example, appearing in the screen. There is nothing 'beneath' it: no behavior whatsoever has been specified for it yet. You now start entering scenarios by clicking and dragging, playing in the user's inputs and the system's responses, indicating whether things are hot or cold, instantiated or generic, and more. The interactive process also includes means for refining the system's structure as you progress, 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 accurately capture the played-in scenarios. Thus, we are automating the construction of rigorous and comprehensive requirements from a

friendly, intuitive and user-oriented play-in capability, rather than employing abstract engineer-oriented languages.

Here too, there is much research still to be done. While the idea of play-in scenarios has a nontrivial mathematical/algorithmic side, a large part of the effort needed is related to its human aspects. There must be powerful, yet natural and easy-to-use means for interacting with an essentially behavior-free 'system shell', in order to tell it what we want from it. A separate paper will be devoted to describing the details of the first version of a play-in environment that I have been developing with a collegue.

The complete system development dream is summarized in Figure 4.

* * *

It is probably no great exaggeration to say that there is a lot more that we *don't* know and *can't* achieve yet in this business than what we do know and can achieve. Besides the topics we have discussed, there are many additional issues that we haven't touched upon here at all, which require lots and lots of further research and development in order to be satisfactorily incorporated into the overall scheme. They include real time analysis, automatic diagram layout, and dealing with hybrid systems that have continuous as well as discrete facets.

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

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 depcited 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 [2] and Larry Constantine and Ed Yourdon [1]. David Parnas' work over the years was very influential too.

In the mid-1980s, several methodology teams enriched this basic SA/SD model by providing a way to add behavior to these efforts, using state diagrams or the richer language of statecharts [3]. These are Ward and Mellor [13], Hatley and Pirbhai [7], and the Statemate team [4]. 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 the modeler 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, etc. — and an engine, and you then 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 [5, 10], 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 [4] (see also http://www.ilogix.com). An updated and detailed summary of the SA/SD languages for structure and behavior, their relationships and the way they are embedded in Statemate appears in [6].

Of course, modelers need not adopt state machines or statecharts to describe behavior. There are many other possible choices, and these could have been linked with the SA/SD diagrams for specifying the structure and data flow in ways similar to those just mentioned. They include such visual formalisms as Petri nets or SDL diagrams, and more algebraic ones like CSP or CCS [11, 12, 8, 9].

[1] Constantine, L. L., and E. Yourdon, *Structured Design*, Prentice-Hall, Englewood Cliffs, 1979.

[2] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

[3] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog.* **8** (1987), 231–274. (Preliminary version appeared as Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, Feb. 1984.)

[4] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Trans. Soft. Eng.* **16** (1990), 403–414. (Preliminary version in *Proc. 10th Int. Conf. Soft. Eng.*, IEEE Press, New York, 1988, pp. 396–406.)

[5] Harel, D., 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.

[6] Harel, D., and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998.

[7] Hatley, D., and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1987.

[8] Hoare, C.A.R., "Communicating Sequential Processes", *Comm. Assoc. Comput. Mach.* **21** (1978), 666–677.

[9] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vo. 92, Springer-Verlag, Berlin, 1980.

[10] Pnueli, A., "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", *Current Trends in Concurrency* (de Bakker et al., eds.), Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, Berlin, 1986, pp. 510–584.

[11] Reisig, W., *Petri Nets: An Introduction*, Springer-Verlag, Berlin, 1985.

[12] SDL: ITU-T Recommendation Z.100, Languages for telecommunications applications: Specification and description language, Geneva, 1999.

[13] Ward, P., and S. Mellor, *Structured Development for Real-Time Systems* (Vols. 1, 2, 3), Yourdon Press, New York, 1985.

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 level to the modeling level and to use visual formalisms. Inspired by entity-relationship (ER) diagrams [2], several methodology teams recommended various forms of class and object diagrams for modeling system structure [1, 3, 7, 9]. To model behavior, most object-oriented modeling approaches adopted statecharts [4]. Each class has an associated statechart, which describes the behavior of any instance object.

In the OOAD world, the issue of connecting structure and behavior is subtler and a lot more complicated than in the SA/SD one. 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 (ROOM) method of [9], and is now part of the Rational RealTime tool (see http://www.rational.com). Another is Rhapsody (see http://www.ilogix.com), which is based on the work of Eran Gery and the present author in [5], on executable object modeling with statecharts.

The work reported on in [5] centers on a carefully constructed language set that includes class/object diagrams adapted from the Booch method [1] and the OMT method [7], driven by statecharts for behavior. This pair of languages also serves as the executable heart of the Unified Modelling Language (UML), put together by a team led by Grady Booch, James Rumbaugh and Ivar Jacobson, which the Object Management Group (OMG) adopted as a standard in 1997 (see http://www.omg.org). The class/object diagrams and the statecharts part of the UML, as described in [5], is called XUML (for executable UML). Thus, XUML is the part of UML that specifies unambiguous, executable, and therefore implementable, models.

The UML has several means for specifying more elaborate aspects of system structure and architecture (for example, packages and components). An important part of the UML for specifying requirements is Jacobson's use cases [6]. A large amount of further information on the UML can be found in the OMG's web site (http://www.omg.org) or in [8].

[1] Booch, G., *Object-Oriented Analysis and Design, with Applications* (2nd edn.), Benjamin-Cummings, 1994.

[2] Chen, P. P., "The Entity-Relationship Model: Toward a Unified View of Data", *ACM Trans. on Database Systems* **1** (1976), 9–36

[3] Cook, S. and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice Hall, New York, 1994.

[4] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog.* **8** (1987), 231–274. (Preliminary version appeared as Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, Feb. 1984.)

[5] Harel, D., and E. Gery, "Executable Object Modeling with Statecharts", *Computer* (July 1997), 31–42.

[6] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley, 1992.

[7] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[8] Rumbaugh, J., I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[9] Selic, B., G. Gullekson and P. T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.

References

[1] Apt, A., *Verification of Sequential and Concurrent Programs* (2nd edn.), Springer Verlag, New York, 1997.

[2] Damm, W., and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Formal Methods in System Design*, to appear. (Preliminary version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems* (*FMOODS'99*), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293–312.)

[3] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog.* **8** (1987), 231–274. (Preliminary version appeared as Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, Feb. 1984.)

[4] Harel, D., "Biting the Silver Bullet: Toward a Brighter Future for System Development", *Computer* (Jan. 1992), 8–20.

[5] Harel, D., and E. Gery, "Executable Object Modeling with Statecharts", *Computer* (July 1997), 31–42.

[6] Harel, D., and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications", *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, Lecture

Notes in Computer Science, Springer-Verlag, 2000, to appear. (Also, Tech. Report MCS99-20, The Weizmann Institute of Science, Rehovot, Israel, Oct. 1999.)

[7] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, ACM Press/Addison-Wesley, 1992.

[8] Manna, Z., and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, New York, 1992.

[9] MSCs: ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.

[10] Pnueli, A., and R. Rosner, "On the Synthesis of a Reactive Module", *Proc. 16th ACM Symp. on Principles of Programming Languages*, Austin, TX, January 1989.

[11] Schlor, R. and W. Damm, "Specification and verification of system-level hardware designs using timing diagrams", *Proc. European Conference on Design Automation*, Paris, France, IEEE Computer Society Press, pp. 518 – 524, 1993.

[12] Unified Modeling Language (UML) documentation, available from the Object Management Group (OMG), http://www.omg.org.