From Play-In Scenarios To Code: An Achievable Dream

(Preliminary version, January 2000)

David Harel*

To appear in *Proc. Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science, Vol. 1783, Springer-Verlag, 2000.

Abstract

We discuss the possibility of a complete system development scheme, supported by semantically rigorous automated tools, within which one can go from an extremely high-level, user-friendly requirement capture method, which we call *play-in scenarios*, to a final implementation. A cyclic process consisting of verification against requirements and synthesis from requirements plays an important part in the scheme, which is not quite as imaginary as it may sound.

Over the years, the main approaches to high-level system modeling have been structured-analysis and structured design (SA/SD), and object-oriented analysis and design (OOAD). The two are about a decade apart in initial conception and evolution. SA/SD started out in the late 1970's by DeMarco, Yourdon and others, and is based on 'lifting' classical procedural programming concepts up to the modeling level and using diagrams for visualization [3, 6]. The result calls for modeling system structure by functional decomposition and the flow of information, depicted by hierarchical data-flow diagrams. As to system behavior, the mid 1980's saw several methodology teams (such as Ward and Mellor [35], Hatley and Pirbhai [17] and our Statemate team [14]) enriching the basic SA/SD model with means for modeling behavior, using state diagrams or the richer language of statecharts [9]. A state diagram or statechart is associated with each function or activity, describing its behavior.[1]

Many nontrivial issues had to be worked out in order to properly connect structure with behavior, enabling the modeler to construct a comprehensive and semantically rigorous model of the system (see Fig. 1); it is not enough to simply decide on a behavioral language and then associate each function or activity with a behavioral description.[2] The above teams struggled

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

[1]Of course, there are many other possible choices for a language to specify behavior, including such visual languages as Petri nets [29] or SDL diagrams [33], and more algebraic ones like CCS [24] and CSP [19].

[2]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...

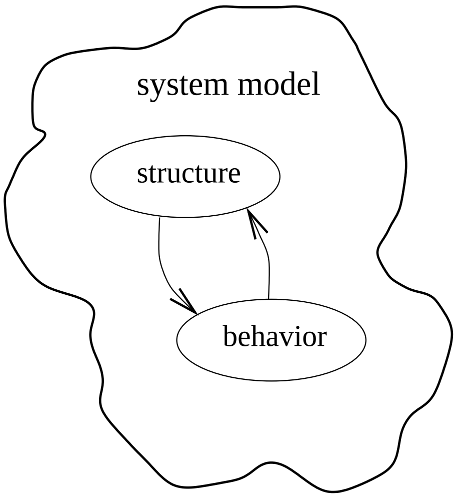

Figure 1: A system model

with this issue, and their decisions on how to link structure with behavior ended up being very similar. Detailed descriptions of the way this is done in the SA/SD framework appear in [35, 17, 16].

Careful behavioral modeling and its close linking with system structure are especially crucial for embedded, reactive systems [15, 26], which constitute the kinds of systems this paper is most concerned with, and of which real-time systems are a special case. The availability of a rigorous semantical basis for the model — notably for the behavioral parts — is what leads to the possibility of executing models and running actual code generated from them. The first commercial tool to enable model execution and code generation from high-level models was Statemate from I-Logix in 1987 (see [14, 20]). We shall have more to say about executability and code generation later on. Here, code need not necessarily be of the kind that results in software; it could be code in a hardware description language, resulting ultimately in real hardware.

Turning to object-orientation, following developments in OO programming, proposals for object-oriented modeling (analysis and design; that is, OOAD) started to show up in the late 1980's. Here too, the basic idea for system structure was to 'lift' concepts from the programming level up to the modeling level, and to use diagrams, or what has been termed *visual formalisms* in [10]. For example, the basic structural model for objects in Booch's method [1], in the OMT method of Rumbaugh and his colleagues [31], in the ROOM method [34], and in many others (e.g., [4]), features a graphical notation for such OO notions as classes and instances, relationships and roles, and aggregation and inheritance. Visuality is achieved by basing this model on an enriched form of entity-relationship diagrams. As to system behavior, most object-oriented modeling approaches, including those just listed, adopted the statecharts language [9] for this (or a variant thereof). A statechart is associated with each class, and its role is to describe the behavior of the instance objects.

The issue of connecting structure and behavior in the OOAD world is more subtle and a lot more complicated than in the SA/SD one. Classes represent dynamically changing collections of concrete objects, and behavioral modeling must address issues related to their creation and destruction, the delegation of messages, the modification and maintenance of relationships, aggregation, inheritance, etc. The test of whether these have been dealt with properly is, of course, whether the inter-links of Fig. 1 are defined sufficiently well to allow model execution and code generation. This has been achieved only in a couple of cases. One is the ObjecTime tool, which is based on the ROOM method of [34], and the other is the Rhapsody tool (see [20]), which is based on the executable object modeling work of [12]. Executable object modeling is a carefully worked out language set based on the class/object diagrams of [1, 31], driven by statecharts for behavior, and addressing the issues above in a rigorous way.

The pair of languages described in [12] also serve as the heart of the recent UML language set [36, 30], which was adopted in 1997 as a standard by the Object Management Group. In fact, we shall refer to this part of the UML, namely the class/object diagrams and the statecharts, as described in [12] or in the UML documents [36, 30], by the term *Base-UML*. Thus, Base-UML is that part of the UML that is used to specify unambiguous, executable, and therefore implementable, models. Perhaps it should be called *Executable-UML*, or simply *XUML* or *X-UML*. In any event, we shall stick to 'Base-UML' here.[3]

Indeed, if we have the ability to generate full code, we would eventually want that code to serve as the basis for the final implementation. Current tools are in fact capable of producing quality code, good enough for the implementation of many kinds of 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 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. The broad arrow on the right-hand side of Fig. 2 is now relevant: with the convention that solid broad arrows denote automated processes, it shows the system model giving rise to automatic generation of the full runnable code.

* * *

Let us now discuss model execution.[4] Why would we want to execute models? Clearly, in order to test and debug them. Against what should we be carrying out the testing? Clearly, against requirements. What kinds of requirements are relevant to high-level system modeling with visual formalisms? Well, requirements are, by their very nature, 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.

[3]The UML has several means for specifying more elaborate aspects of the structure and architecture of the system under development (for example, packages and components), but we shall not get into these here. Our emphasis is on classical object-oriented structure and full behavior.

[4]See also [11] for a more detailed discussion of various kinds of model execution that one might desire.

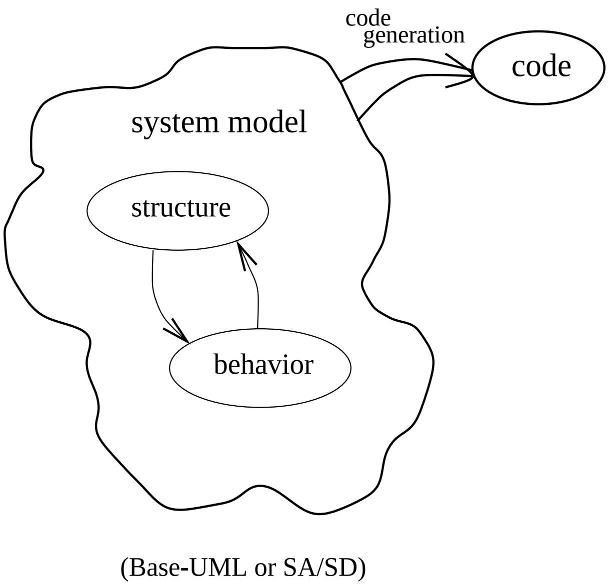

Figure 2: System modeling with full code generation

Since in this paper we concentrate on processes that can be automated, we shall not discuss informal requirements, written in natural language or pseudo code. Instead, we concentrate on rigorous, precisely defined 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 early work of Floyd and Hoare on invariant assertions and termination statements [8, 18], and in the many variants of temporal logic [26, 22, 23]. These make it possible to express the two main kinds of requirements of interest: safety properties (a bad thing can't happen; e.g., this program will never terminate with the wrong answer, or this elevator door will never open between floors), and liveness properties (good things must happen; e.g., 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*). This graphical language was adopted long ago as a standard by the CCITT telecommunication organization, currently the ITU (see [25]), and in the UML it manifests itself — in a slightly weaker way – as the language of *sequence diagrams*. 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 Jacobson's methodological notion of *use-cases* [21]: In the early stages of system development, engineers typically come up with use-cases, and then proceed to 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 linear or quasi-linear fashion, in terms of temporal progress.[5] In other words, we are specifying the scenarios, or the 'stories',

[5]We mention tasks and processes here, since although we couch much of our discussion in the terminology of

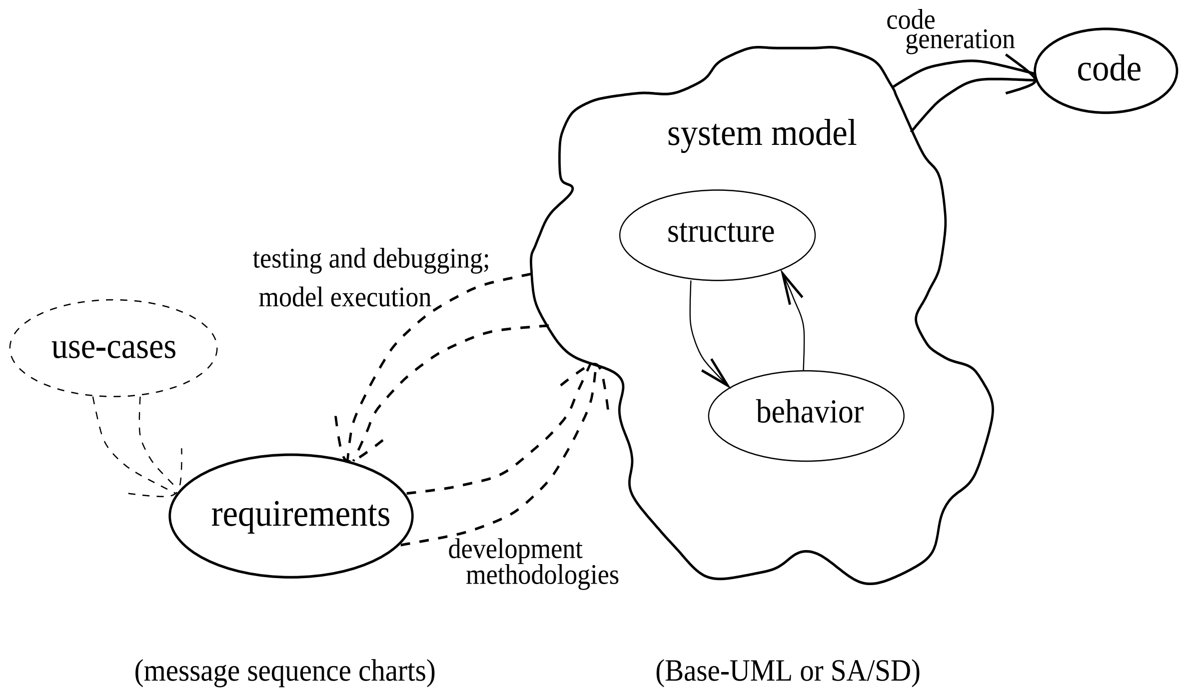

Figure 3: System modeling with 'soft' links to requirements

that the final system should, and hopefully will, satisfy and support. See Fig. 3.

However, it is important to realize that these scenarios are not part of the system. They are part of the requirements *from* the system. They are constructed (often based on the less detailed and more abstract use-cases) in order to capture the scenarios that we hope, desire, and want our system to satisfy, when implemented. Of course, one can ask why these scenarios cannot be part of the implementable system model. We shall return to this question soon, but we should first contrast this int*er*-object 'one-story-for-all-objects' approach with the dual int*ra*-object 'all-stories-for-one-object' approach manifest in the Base-UML 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 of the (tasks or processes or) object instances, providing details of its behavior under all possible conditions and in all possible 'stories'. This intra-object specification is what we need as an output from the design stage, since it is directly implementable (ultimately, the final software will consist of code specified for each object). It is at the heart of the system model of Figs. 1 and 2, which must support, or satisfy, the scenarios as specified in the MSCs. The MSCs themselves cannot be implemented; only the model can.[6] MSCs and sequence diagrams are thus the requirements,

object-orientation and the UML, there is nothing special to objects in what we are discussing.

[6]You can't simply prepare a collection of 1000 scenarios and call that your system. How would it operate? What would it do under general dynamic circumstances? How are these scenarios related? What happens if things occur that simply do not fall under any of the scenarios? And on and on.

whereas Base-UML provides the implementable model.

* * *

Let us now turn to the broad dashed arrows in Fig. 3, those between the requirements and the system model. Going from the requirements to the model is another long-studied issue, and many system development methodologies provide guidelines, heuristics, and sometimes carefully worked-out step-by-step processes for this. The reason this left-to-right arrow is dashed is obvious: however good and useful, such processes are 'soft' methodological recommendations as to how to proceed, and are not rigorous and automated.

The arrow going from the system model to the requirements depicts the testing and debugging of the model against the requirements, using, e.g., model execution. Here is one of the neat ways this can be done, as supported by the Rhapsody tool. The user specifies the requirements as a set of sequence diagrams (perhaps instantiating previously prepared use-cases), and then puts them aside for the moment. Suppose this results in a diagram called A. Later, when the system model has been specified, it can be executed[7] and Rhapsody can be instructed to automatically construct, on the fly, animated sequence diagrams that show the dynamics of object interaction as they actually happen during execution. Suppose this results in diagram B. Upon completion of the execution, Rhapsody can be asked to compare diagrams A and B, and it will highlight any inconsistencies, thus helping debug the behavior of the system against the requirements. While this is a powerful and extremely useful way of working, we must remember that it is really limited to those executions of the model that we actually carry out, and is thus akin to classical testing and debugging. Since in general there will be infinitely many inputs or runs of the system, there could always be some out there that were not checked, and which could violate the requirements by being inconsistent with, e.g., the sequence chart A. As Dijkstra famously put it years ago, "testing and debugging cannot be used to demonstrate the absence of errors, only their presence" [7]. This 'softness' of the debugging process is the reason this arrow is dashed too. So much for Fig. 3.

Two points must now be made regarding MSCs and sequence diagrams. The first is one of exposition: by and large, the true role of these 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 something like "sequence diagrams can be used to specify behavior", and later it might say that "statecharts can be used to specify behavior". Sadly, the reader isn't told anything about the fundamental difference between the two — their different roles and indeed 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 UML and the incoherent recommendations as to what it means to 'specify' a system.

[7]Rhapsody actually executes its Base-UML models by generating code from them and running the code in a way that is linked to the visual model.

The second point is more substantial. As a requirements language, all known versions of MSCs, including the ITU standard [25] and the sequence diagrams adopted in the UML [36, 30], 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 *may 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 (i.e., 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....[8] Another troublesome drawback of MSCs is their inability to specify 'no-go' scenarios (or, as we may call them, *anti-scenarios*), which are crucial in setting up safety requirements. These are scenarios whose occurrence we want to forbid; they are to be explicitly *dis*allowed. In short, there is a serious need for a more powerful language for sequences.

In a recent paper with Damm [5], this need has been addressed, by proposing an extension of MSCs, called *live sequence charts* (or *LSCs*). One of the main extensions deals 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 elements, the *hot* ones, as we call them in [5], make it possible to specify anti-scenarios too. LSCs also support subcharts, synchronization, branching and iteration. It is not clear yet whether this language is exactly what is needed, and a lot more work on it is definitely required. Experience using it must be gained and an implementation is badly needed. But the proposal is there, and LSCs are a preliminary candidate for a far more powerful way of visually specifying behavioral requirements of a system model, and thus of the system's final implementation.

* * *

Since their expressive power is far greater than MSCs (it is essentially that of Base-UML itself), LSCs also make it possible to start looking more seriously at this 'grand dichotomy' of reactive behavior, namely, the relationship between the aforementioned dual views of behavioral description — the inter-object requirements view captured by LSCs and the intra-object implementable view captured by statecharts in Base-UML. Let us try to do so, by referring to Fig. 4.

The difference between Figs. 3 and 4 is in the two arrows between the requirements and the model, which have been 'un-dashed'. 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 (e.g., Base-UML) and the requirements (e.g., LSCs).[9]

[8] Usually, however, there is a minimal, often implicit, requirement that there should be at least one run of the system that winds its way correctly through any specificed sequence chart.

[9] Requirements could equally well have been specified using some powerful form of temporal logic with path quantifiers; see, e.g., [22, 23] or certain kinds of timing diagrams [32].

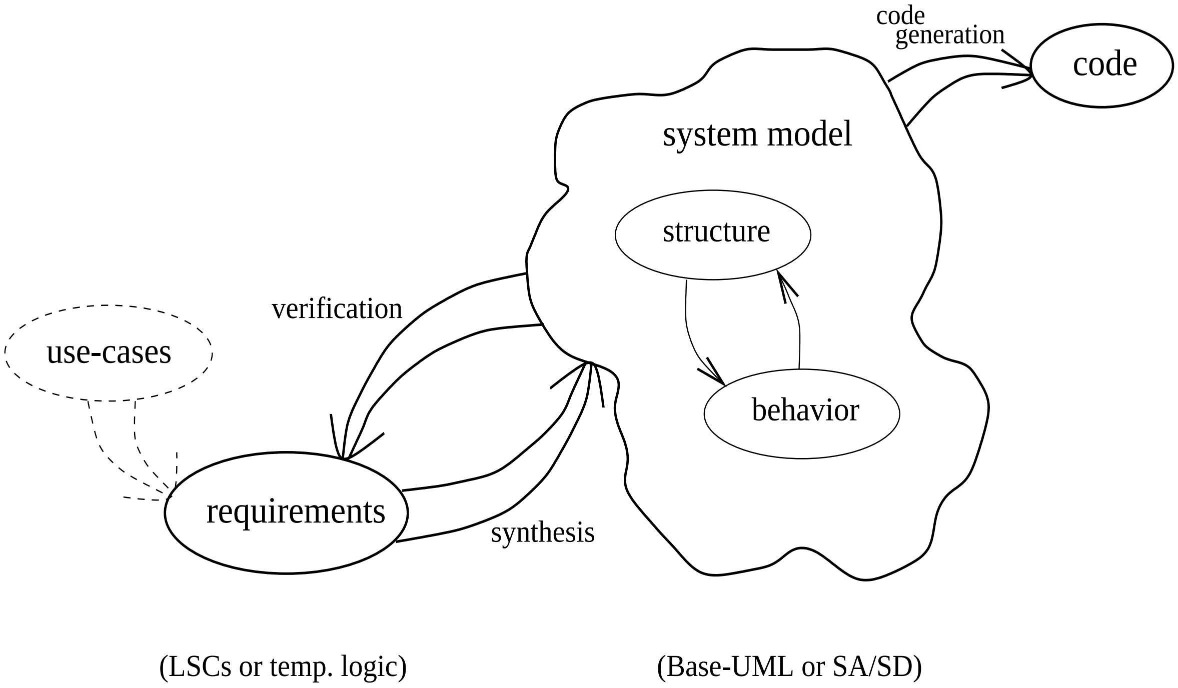

Figure 4: System modeling with 'hard' links to requirements

Going from right to left, instead of testing and debugging by executing models, we are now 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 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 specifies as *not* allowed to happen, will indeed never happen, and things it specifies as *having* to happen (and/or having to happen within certain time constraints), will indeed happen — facts not verifiable in general by any amount of execution. And it means a lot more too. This paper is not a treatise on verification, so we shall say no more about this here, except to note that although verification in general constitutes a non-computable algorithmic problem, ever since the early work of Floyd and Hoare [8, 18], and through the work on temporal logic [23] and model checking verification [2], rigorously verifying programs and systems — hardware and software — has matured, and has become more and more viable. These days we can safely say that it *can* be carried out in many, many cases, especially in the finite-state ones that arise in the realm of reactive, real-time systems. Work is in progress these days on enriching the Statemate tool [14, 16, 20] with true verification capabilities. Doing the same for an OOAD tool like Rhapsody or Rose-RT is just a matter of time. Before long, I believe, we will be routinely using such tools to verify models agains requirements.

In the opposite direction, from the requirements to the model, 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 carry out true *synthesis* directly from those dreams and hopes, if they are indeed implementable. We want to be able to automatically generate a system model (say, statecharts) from the requirements (say, LSCs). 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 and the statechart style 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.

How can we synthesize a good first approximation of the statecharts from the LSCs? Several researchers have addressed this issue in the past, including work on certain kinds of synthesis from temporal logic [27, 28] and timing diagrams [32]. More recently, in [13], we have been able to present a first-cut attempt at algorithms for synthesizing state-machines and statecharts from LSCs (albeit, in a slightly restricted setup, and resulting in models that could become too large to work with). This is done by first determining whether the requirements are consistent (i.e., whether there is *any* system model satisfying them), and then using the proof of consistency to synthesize a model. There is still a lot of rather deep research to be done here, and work is in progress as we write. I believe this problem will eventually end up like verification — hard in principle, but not beyond a practical and useful solution.

What does all this mean? Well, it is tempting to say that if we have the picture set up as in Fig. 4, we don't even need verification or testing: just go directly from left to right. State your requirements, have one part of your tool synthesize the system model and another part generate code, and you are all set. Again, this in not a treatise on incremental development of systems (although the more ambitious parts of this paper implicitly suggest that the classical life cycle models might eventually have to be modified somewhat), but obviously one would want to go through a cycle of development phases, producing continuously refined and extended versions of the system. This cycle, we suggest, would repeatedly and incrementally follow some combination of the dashed arrows of Fig. 3 — development methodologies and testing and debugging — and the solid ones of Fig. 4 — synthesis and verification — in both horizontal growth of the system under development and in its vertical refinement. This will have to be done based not on quick-and dirty methods written up hastily in shallow methodology books, but on the deep and profound wisdom that will have to be accumulated over years of experience using these techniques. It will not happen overnight, even if all the required tools were just around the corner.

* * *

To complete the dream this paper has tried to sketch, albeit superficially, I would like to introduce one additional idea, that of *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 especially transparent and impressive (useful too) when you work with a soft panel mock-up of the system's final interface or even a hard version of the system's actual hardware, as is possible in Statemate and Rhapsody. You can play-out a scenario by standing in, so to speak, for the system's environment, introducing events and changes in values, etc., and observing the results as they unfold (see [11]). What is proposed here is to play-*in* scenarios in order to set up the requirements, perhaps driven by use-cases. This will be done by working directly opposite such a mock-up of the system's interface (think of a cell-phone, for example), using a highly user-friendly method of 'teaching' your tool about the desired and undesired scenarios. The interactive process will also include means for refining the system's structure as you progress, e.g., forming composite objects and their aggregates and setting up inheriting objects, all reflected in a modified mock-up interface. As the process of playing in the scenario-based requirements continues, the underlying tool will automatically and incrementally generate LSCs (not merely MSCs) that are consistent with these 'teachings'. Thus we are automating the construction of rigorous and comprehensive requirements from a friendly, intuitive and natural play-in capability, which could even be carried out by the customer.

Here too, there is much research still to be done. While there is a nontrivial mathematical/algorithmic side to this too, we must deal with the human aspect: we have to find 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. We are currently at work on this rather exciting possibility, and hope to be able to report pretty soon on an initial proposal for play-in scenarios and a first-cut prototype implementation of such a capability.

Fig. 5 is a schematic attempt to summarize the story as a grand dream. In the figure, we have also included a second broad arrow in the right-hand upper part. It indicates the ability of the user to 'round-trip' back from the code to the model: making changes in the former reflects automatically back as changes in the visual formalisms of the latter. 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 ability too will become commonplace in the future, and that the techniques enabling it will become more powerful and far broader in applicability.

* * *

In summary, 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. The efforts of scores of researchers, methodologists and language designers have resulted in a lot more than we could have hoped for ten or twenty years ago, and for this we should be thankful and humble. Still, I maintain that 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 to us yet — but one that is not

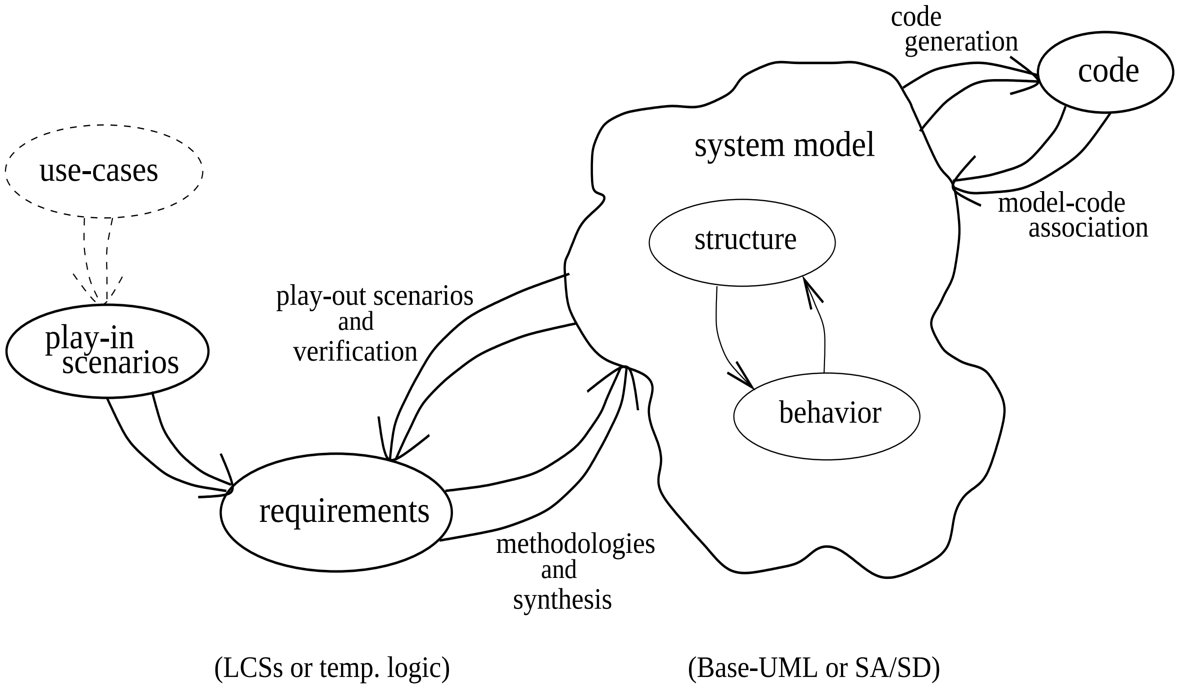

Figure 5: The dream in full

unattainable. If and when it comes true, it could have a significant effect on the way complex systems are developed.

References

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

[2] Clarke, E.M., O. Grumberg and D. Peled, *Model Checking*, Mit Press, 1999.

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

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

[5] Damm, W., and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *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.

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

[7] Dijkstra, E.W., "Notes on Structured Programming", in *Structured Programming*, Academic Press, New York, 1972.

[8] Floyd, R.W., "Assigning Meanings to Programs", *Proc. Symp. on Applied Math.*, (Vol. 19: "Mathematical Aspects of Computer Science"), American Math. Soc., Providence, RI, pp. 19–32, 1967.

[9] 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.)

[10] Harel,D., "On Visual Formalisms", *Comm. Assoc. Comput. Mach.* **31**:5 (1988), 514–530.

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

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

[13] Harel, D., and H. Kugler, "Synthesizing Object Systems from Live Sequence Charts", submitted for publication, 1999.

[14] 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.)

[15] 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.

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

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

[18] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", *Comm. Assoc. Comput. Mach.* **12** (1969), 576–583.

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

[20] I-Logix, Inc., products web page, http://www.ilogix.com/fs_prod.htm.

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

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

[23] Manna, Z., and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*, Springer-Verlag, New York, 1995.

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

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

[26] 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.

[27] 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.

[28] Pnueli, A., and R. Rosner, "On the Synthesis of an Asynchronous Reactive Module", *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, Berlin, 1989, pp. 652–671.

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

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

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

[32] 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.

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

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

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

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