

On the Behavior of Complex Object-Oriented Systems*

David Harel

The Weizman Institute of Science, Rehovot, Israel

and

I-Logix, Inc., Andover, MA

Over the years, the main approaches to high-level system modeling have been structured-analysis and object-orientation (OO). The two are about a decade apart in initial conception and evolution. SA started out in the late 1970's by De Marco, Yourdon and others, and is based on "lifted" classical procedural programming concepts up to the modeling level and using diagrams [CY79]. 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/Mellor [WM85], Hatley/Pribhai [HP87] and our own Statemate team [HHN⁺90]) making detailed recommendations enriching the basic SA model with means for capturing behavior based on state diagrams, or the richer language of statecharts [Har87]. A state diagram or statechart is associated with each function to describe its behavior. Carefully defined behavioral modeling, we should add, is especially crucial for embedded, reactive, and real-time systems. A detailed description of the way this is done in the SA framework appears in [HM98]. The first tool to enable model executability and code synthesis of high-level models was Statemate, made commercially available in 1987 (see [HHN⁺90], [Inc]).

OO modeling started in the late 1980's. Here too, the basic idea for system structure was to "lift" concepts from object-oriented programming up to the modeling level, and to do things with diagrams. Thus, the basic structural model for objects in Booch's method [Boo94], in OMT [RBP⁺91], in the ROOM method [SGW94], and in many others (e.g., [CD94]), has notation for 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 systems behavior, most OO modeling approaches, including those just listed, adopted the statechart language for this. A statechart is associated with each class, and its role is to describe the behavior of the instance objects.

However, here are subtle and complicated connections between structure and behavior, that do not show up in the simpler SA paradigm. Here classes represent dynamically changed collections of concrete objects, and behavioral modeling must address issues related to their creation and destructions, the delegation of messages, the modification and maintenance of relationships, aggregation, true inheritance, etc. These issues were treated by OO methodologists in a broad spectrum of degrees in detail — from vastly insufficient to adequate. The test, of course, is whether the languages for structure and behavior and their

*Reprint of abstract from *OMER Workshop Proceedings*, Peter Hofmann and Andy Schüerr (eds.), Bericht Nr. 1999-01, University of the Federal Armed Forces Munich, Neubiberg, Germany

inter-links are defined sufficiently well to allow full model execution and code synthesis. This has been achieved only in a couple of cases, namely in the ObjecTime tool (based on the ROOM method of [SGW94]), and in the Rhapsody tool. Rhapsody (see [Inc]) is based on the executable modeling work presented in [HG97], which was originally intended as a carefully worked out language set based on Booch and OMT object model diagrams driven by statecharts, and addressing the issues above in a way sufficient to lead to executability and full code synthesis.

In a remarkable departure from the similarity in evolution between the SA and OO paradigms for system modeling, the last three years have seen OO methodologists working together. They have compared notes, have debated the issues, and have finally cooperated in formulating the UML, which was adopted in 1997 as a standard by the OMG (see [Cor]). This sweeping effort, which in its teamwork is reminiscent of the Algol'60 and Ada efforts, has taken place under the auspices of Rational Corp., spearheaded by Booch, Rumbough and Jacobson. Version 0.8 of the UML was released in 1996 and was a rather open-ended and vague, lacking in detail and well thought-out semantics. For about a year, the UML team went into overdrive, with a lot of help from methodologists and language designers from outside Rational Corp. Our team contributed quite a bit too, and the languages underlying Rhapsody [HG97], [Inc] are indeed the executable kernel of the UML. The version of the UML adopted by the OMG is thus much tighter and more solid than version 0.8. With some more work there is a good chance that the UML will become not just an officially approved standard, but the main modeling mechanism for the software that is constructed according to the object-oriented doctrine. And this is no small matter, as more and more software engineers are now claiming that more and more kinds of software are best developed in an OO fashion.

The recent wave of popularity that the UML is enjoying will bring with it not only the UML books written by Rational Corp. authors (see, e.g., [RJB99]), but a true flood of books, papers, reports, seminars, and tools, describing, utilizing, and elaborating upon the UML, or purporting to do so. Readers will have to be extra-careful in finding the really worthy trees in this forest. Despite this, one must remember that right now UML is a little *too* massive. We understand well only parts of it; the definition of other parts has yet to be carried out in sufficient depth as to make clear their relationships with the constructive core of UML (the class diagrams and the statecharts). Moreover, there are still major problems in the general area of behavioral specification and design of complex object-oriented systems that await treatment. These still require extensive research.

Here are brief discussions of two examples of research directions that seem to me to be extremely important. One has to do with message sequence charts (MSC's) and their relationship with state-based specification, and the other has to do with inheriting behavior.

As to the first one, there is a dire need for a highly expressive MSC language, with a clearly defined graphical syntax and a fully worked out formal semantics. Such a language is needed in order to construct semantically meaningful computerized tools for describing and analyzing use-cases and scenarios. It is also a prerequisite to a thorough investigation of what might be the problem in object-oriented specification. The former is what engineers will typically do in the early stages of behavioral modeling; namely, they come up with use-cases and the scenarios that capture them, specifying the inter-relationships

between the processes and object instances in a linear or quasi-linear fashion of temporal progress. That is, they come up with the description of the scenarios, or “stories” that the system will support, each one involving all the relevant instances. A language for scenarios is best used for this. The latter, on the other hand, is what we would like the final stages of behavioral modeling to end up with; namely, a complete description of the behavior of each of the instances under all possible conditions and in all possible “stories”. For this, a state-machine language such as statecharts appears to be most useful. The reason the state-machine intra-object model is what we want as an output from design stage is for implementation purposes: ultimately, the final software will consist of code for each process or object. These pieces of code, one for each process or object instance, must — together — support the scenarios as specified in the MSC’s. Thus the “all relevant parts of the stories for one object” descriptions must implement the “one story for all relevant objects” descriptions.

Now, there are several versions of MSC’s, including the ITU standard (see [ITU96]), and the UML also has a version of sequence diagrams as part of its language. However, both versions are extremely weak in expressive power, being based essentially on simple constraints on the partial order of events. Nothing much can be specified about what the system will actually do when run. A particular troublesome issue is the need to be able to specify “no-go” scenarios, ones that are not allowed to occur. In short, there is a serious need for a more powerful language for sequences. In a recent paper [DH99], we have addressed this need, proposing an extension of MSC’s, which we call *live sequence charts* (or *LSCs*). One of the main extensions deals with specifying “liveness”, i.e., things that must occur. LSCs allow 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. (In doing so it makes possible the natural specification of forbidden behavior.) LSCs also support subcharts, synchronization, branching and iteration. It is far from clear whether this language is exactly what is needed: more work on it is required, experience in working with it, and of course an implementation. Nevertheless, it does make it possible to start looking seriously at the two-way relationship between the aforementioned dual views of behavioral description. How to address this *grand dichotomy of reactive behavior*, as we like to call it, is a major problem. For example, how can we synthesize a good first approximation of the statecharts from LSCs? Finding efficient ways to do this would constitute a significant advance in the automation and reliability of system development. In very recent work, as of yet unpublished, we propose a first-cut at devising such synthesis algorithms and at analyzing their complexity [HK99a].

The second direction of research involves inheriting behavior. Inheritance is one of the key topics in the object-oriented paradigm, but when working with analysis and design levels (rather than in the programming stage) it is not at all clear what exactly it means for an object of type *B* to be also an object of the more general type *A*. In virtually all approaches to inheritance in the literature, the **is-a** relationship between classes *A* and *B* entails a basic minimal requirement of *protocol conformity*, or subtyping, which roughly means that it should be possible to “plug-in” a *B* wherever an *A* could have been used, by requiring that what can be requested of *B* is consistent with that can be requested of *A*.

In addition, *structural conformity*, or subclassing, is often requested, to the effect that *B*'s internal structure, such as its set of composites and aggregates, is consistent with that of *A*. Nevertheless, these form only weak kinds of subtyping, and they say little about the *behavioral conformity* of *A* and *B*. They require only that the plugging in be possible without causing incompatibility, but nothing is guaranteed about the way *B* will actually operate when it replaces *A*. Thus we don't have full behavioral substitutability, but merely a form of consistency. In fact, *B*'s response to an event or an operation invocation might be totally different from *A*'s. Here we are concerned with investigating the plausibility (and indeed also the very wisdom) of guaranteeing full behavioral conformity. In practice, behavioral conformity is often too stringent; many times one does not expect the inheritance relationship between *A* and *B* to mean that anything *A* can do *B* can do and in the very same way. They are often satisfied with guaranteeing that anything *A* can do, *B* can be *asked* to do, and will look like it is doing, but it might do so differently and produce different results.

In recent work, also not yet published [HK99b], we have obtained preliminary results that show that on a suitable schematic, propositional-like level of discourse there are strong connections between questions of inheritance and well-known semantic notions of refinement between specifications (such as trace containment and simulation). We also have several results about the computational complexity of detecting and enforcing behavioral conformity. However, here too there is still much research to be done, including the discovery of restrictions on behavioral specification that would guarantee behavioral conformity, and algorithms for finding out if given models satisfy such restrictions.

Many other significant challenges remain, for which only the surface has been scratched. Examples include true formal verification of software modeled using high-level visual formalisms, automatic eye-pleasing and structure-enhancing layout of the diagrams in such formalisms, satisfactory ways of dealing with hybrid object-oriented systems that involve discrete as well as continuous parts, and much more.

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 business than what we do know and can achieve. Still, the efforts of scores of researchers, methodologists and language designers have resulted in a lot more than we could have hoped ten years ago, and for this we should be thankful and humble.

References

- [Boo94] G. Booch. *Object-Oriented Analysis and Design, with Applications*. Benjamin/Cummings, 2nd edition, 1994.
- [CD94] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling With Syntropy*. Prentice Hall, New York, 1994.
- [Cor] Rational Corp. document on the UML. <http://www.rational.com/uml/index.html>.
- [CY79] L.L. Constantine and E. Yourdon. *Structured Design*. Prentice Hall, Englewood Cliffs, 1979.

- [DH99] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. 3er IFIP Conf. on Formal Methods for Open Object-based Distributed Systems*, pages 293–312. Kluwer Academic Publishers, 1999.
- [Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Prog.*, 8:231–274, 1987.
- [HG97] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, pages 31–42, July 1997.
- [HHN⁺90] D. Harel, 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.*, 16:396–406, 1990.
- [HK99a] D. Harel and H. Kugler. manuscript in preparation. 1999.
- [HK99b] D. Harel and O. Kupferman. manuscript in preparation. 1999.
- [HM98] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.
- [HP87] D. Hatley and I. Pribhai. *Strategies for Real-Time System Specification*. Dorset House, New York, 1987.
- [Inc] I-Logix Inc. products web page. http://www.ilogix.com/fs_prod.htm.
- [ITU96] ITU, Geneva. *ITU-TS Recommendation Z.120: Message Sequence Charts (MSC)*, 1996.
- [RBP⁺91] J. Rumbough, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
- [RJB99] J. Rumbough, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.
- [WM85] P. Ward and S. Mellor. *Structured Development fro Real-Time Systems*, volume 1, 2, 3. Yourdon Press, New York, 1985.