Modeling Complex Behavior Simply
or
How Much Is Too Much?

Stephen J. Mellor
Project Technology, Inc.

http://www.projtech.com
+1-(520) 544-2881
steve@projtech.com

In which we describe how to speed time-to-market in real-time and embedded systems' software development by modeling complex behavior simply. We achieve this goal by working from known facts, expressing behavioral requirements once, improving organizational accessibility, and simplifying notation.

1. The Problem

There is no doubt that the problems we are tackling today are bigger and more complex than ever before. Many products have significant software components, so to bring a product to market quickly, we must develop complex software quickly too. The analysis of such software generally involves modeling desired behavior, and, as usual, there are several ways to do this, some faster than others. To bring complex software to market quickly, we have to make steady, *verifiable progress* during the project; to express *behavioral requirements in one place* so we can easily find them and change them when required; to express our solutions with an *accessible organization*; and to do it all with a *simple but expressive* notation. Let's take a look at each.

Verifiable Progress. How do you get five elephants into a small car? That's easy: put two in the front and three in the back.

In this example, we have taken a difficult problem and broken it into two smaller problems, each of which can then be addressed separately. Lamentably, even after we have broken the unfortunate elephants down into their component parts, we still cannot fit them into the car—though we may well have been able to produce some stunning progress reports along the way.

The problem here is that the progress towards a solution is illusory because there is no way to verify that a particular decomposition makes sense. Worse, there are no termination criteria–we don't know when to stop decomposing.[1]

[1] The problem, that is.

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

To make steady, verifiable progress during the analysis of a problem, we need verifiable facts. In this context, a 'fact' is a particular behavioral requirement. As the analysis proceeds, we propose more and facts, verifying each one as we go, until we have all the facts about the system. In this manner, we can verify that we are making steady progress.

To speed time-to-market, begin with a top-down partitioning of the problem that guides you and your team into the areas that need to be understood. Then spend the remaining 99% of your time on the project working bottom-up. The facts you discover will tell you if you need to adjust the partitioning.

One Fact in One Place. The minimal expression of a problem is one in which each fact–each behavioral requirement–is stated once only. To achieve this, each fact must be a single non-decomposable unit (i.e., *atomic*), and each fact must be independent of all other facts (i.e. *orthogonal*). Clearly, behavioral requirements interact, but the manner of each interaction is itself another fact. For example, we can describe the speed and direction of an airplane as if it were flying through still air, and separately, we can describe the speed and direction of an air mass. The interaction between the two is a third fact: the airplane has some vector relative to that of the air mass. When all the behaviors are separated in this fashion (i.e., they are atomic and orthogonal), we say that the behavior has been *normalized*. Of course, at physical design time, behavior may be reorganized and even duplicated so as to meet performance requirements.

There are several benefits to this approach. First, each fact can be separately verified. Second, it is always faster to do write something once, rather than to write it twice. Third, saying something once avoids contradictions. Fourth, when the time comes, it is significantly easier to change one statement than to find all the statements, and modify each one–and to do so with absolute consistency. All of these features reduce complexity and speed time-to-market.

Accessible Organization. Consider a very large sheet of paper that contains all the pertinent facts. Because each fact is atomic and orthogonal with respect to the others, we know that this is a minimal and non-redundant expression of the problem. However, each fact may be placed on this large sheet of paper in any way, in any order. While the expression is minimal, it is unlikely to be accessible or easy to read.

There needs to be some kind of organization that will make it easier for the reader to find and absorb information. The organization scheme, then, acts as an index, and a re-organization of the model should not affect its content. Therefore, the choice of indexing scheme should add no semantic content.

There are many possible organizations. We could organize the facts alphabetically according to the subject of the sentence. Or we could group facts into clusters based on the conceptual entities involved. If there are many such entities, the entities themselves can be organized in some manner, yielding a hierarchy.

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

A readable organization that contains no semantic content in the index speeds time-to-market through faster understanding of the models.

Simple but Expressive. Since its adoption by the Object Management Group, the Unified Modeling Language (UML) has become a de facto standard notation for expressing behavior. The primary behavior-modeling portion of the UML is based on State Charts. We show an example in Figure 1, drawn from the UML Notation Guide [1].

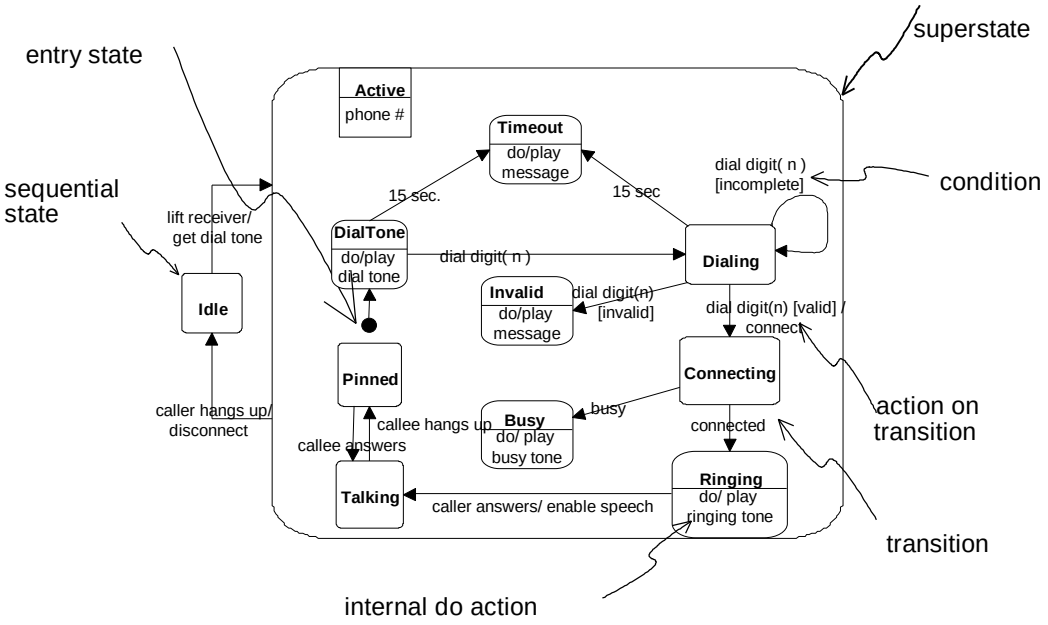


Figure 1: An annotated UML example of a telephone.
The annotations name the various elements shown.

We have not shown all the behavior modeling techniques of the UML, but as you see from the figure, the UML is a rich modeling language, richer than we require to model behavior simply. It is possible to define an equally powerful subset of the standard UML notation that speeds time-to-market through faster learning of the notation.

2. Object-Oriented Partitioning

The deep structure of a subject matter is revealed in its behaviors. When we understand the different behaviors–and encapsulate those behaviors as objects–we can understand the problem as a collection of interacting and related behaviors. The abbreviated model of the telephone shown in Figure 1 can make one outgoing call at a time. Using this as an example, how can we decompose this problem into objects?

To produce a model with normalized behavior, begin by abstracting conceptual entities based on *both* common data and behavior. Construct a separate conceptual entity for each behavior that requires the same data or responds to events in the same way. The

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

abstraction process fundamentally takes into account the behavior: If two instances of a conceptual entity have the same data, but different behavior, we must model them as two conceptual entities. Henceforth, we shall refer to such abstracted conceptual entities as objects, with the understanding that each object represents a typical, but unspecified, instance.

The rules for correct object abstraction follow.

Data must apply to all instances. When some instances of a putative object have data that cannot apply to all instances, abstract another object. For example, the accumulated digits for a call cannot describe all phones, only those making a call, so we should distinguish the concept of a physical telephone from that of a call.

Behavior must apply to all instances. When some instances of a putative object have behavior that cannot apply to all instances, abstract another object. Hence, if some phones behave differently, distinguish between the several different kinds. For example, an off-hook phone has several quite different sets of behaviors, such as making a call or interacting with yet another infernal voice-mail system.

Relationships must apply to all instances. When some instances of a putative object have relationships that only apply to some instances, abstract another object. For example, all physical phones are not all participating in a call, only those phones that are off-hook. This suggests that we distinguish between a call and an off-hook phone.

Further guidelines, rather than rules, follow.

Stimulus causes duplicated response. When a stimulus causes the same response over a range of states, abstract those states as a new object. Applying this principle suggests the separation of the physical telephone from the off-hook phone: When the user hangs up the phone, the result is the same regardless of what is currently being done. It is useful to model those states of the object that have this common response as a separate object, so an off-hook phone acts to encapsulate the behaviors associated with using the phone for its various purposes, including the call or programming an answering machine.

Duplicated sequences of behavior. When an instance can be in two states at once–or there is a seeming requirement to duplicate states–you probably have more than one object. This could occur when modeling multiple calls on the same phone. It will appear that the phone can have two states at once, and it will appear that there are states that are duplicated for each call. Partitioning to separate the call from the physical phone allows there to be two instances of the Call object without two sets of states. The model for the call will show the behavior of one instance of a call.

The result of following these rules may lead you to abstract objects for the telephone as shown in Figure 2.

© Copyright 1997 by Project Technology, Inc.
All rights reserved.



Figure 2: UML abbreviated Class Diagram for the phone.

The Call object encapsulates the data and behavior required actually to carry out a call; the Off-Hook Phone encapsulates the data and behavior off-hook phone as it makes a call (or carries out other activities such as programming an answering machine); the Physical Telephone object encapsulates the data and behavior of the physical device, regardless of its purpose.

This model shows the objects in the problem. Each object has a dynamic model that describes the behavior of one instance. Hence, the model that we build for the telephone, say, describes the behavior of just one phone, and by design, every instance phone has the same behavior even though each phone may be in a different state.

Sometimes, these abstractions will be discovered when modeling behavior, but, with practice, you can do it while abstracting the objects.

3. Modeling and Composing Behavior

We may now model the behavior of each of the three separate objects.

For easy comparison with Figure 1, we show first the behavior of the call and the combination of the Physical telephone and the Off-Hook Phone.

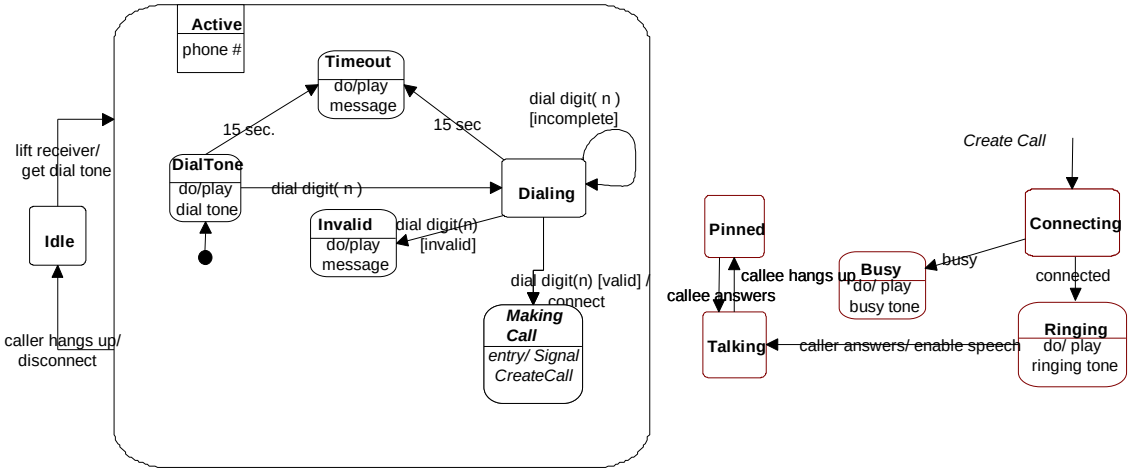


Figure 3: A partitioned model.

Note the creation and receipt of the asynchronous event *Create Call* shown in italics. The model of the Physical Phone/Off-Hook Phone shows a signaled *asynchronous* event: Create Call. The result of this event in the receiver is to create an instance of the call. This is a simple communication between the two objects, shown in Figure 4.

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

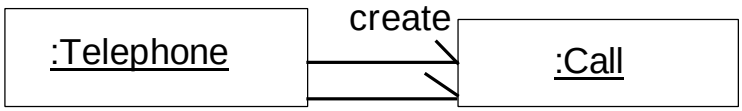


Figure 4: The communication model. The underlined names indicate instances

To complete the model for all three objects, and separate the Physical Telephone from the Off-Hook Phone, use *synchronous* communication to delete the Off-Hook Phone, as shown in Figure 5.

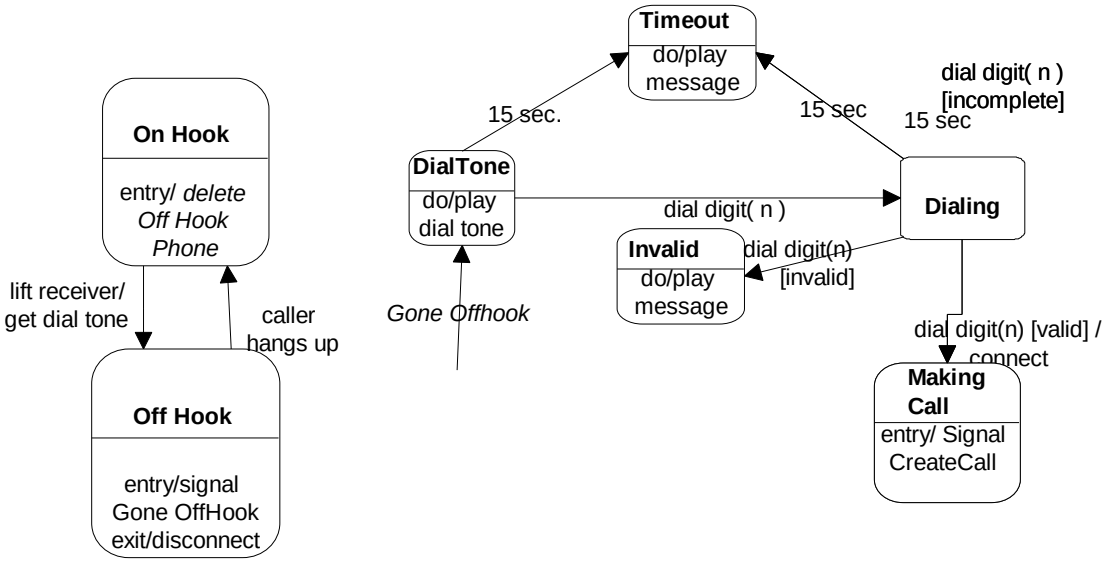


Figure 5: A completely partitioned model.
Note the synchronous communication in the *On Hook* state that deletes an instance

This latter model has the effect of removing the hierarchical state which then leaves all the meaning of the model exposed at a single level of granularity.

4. Building an Index

The example model is rather small, and there is little need to build an index for it, but most models are considerably larger. Surprisingly, developers can deal with a relatively large number of objects at once–if each is reasonably simple–in some cases as many as forty or sixty. However, at some point the model will be large enough that you will need to build an index.

One effective approach is to build a hierarchy of subsystems. The higher (non-substantive) levels show subsystems, and the lowest levels (the leaves of the tree) show only substance–the objects. Build two parallel diagrams with the same partitioning: one

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

showing groups of classes and the associations that span subsystems, and another to show the object communication and the communications that span subsystems.

Note that the resulting hierarchy does not necessarily correspond to a top-down partitioning of the problem. Rather the hierarchy is a bottom-up presentation of, and index to, the objects.

Note also that this view avoids problems with containment. Consider first the original telephone model. The initial view of this model shows a call contained within the telephone. Now consider a more realistic model in which we distinguish between a telephone that initiates a call and one that receives a call. Which object should contain the Call? Should it be the initiator or the recipient of the Call? The answer, of course, is Yes. Building a simple model avoids this question—the placement of the call object is purely a matter of readability.

5. Subsetting the Notation

The UML allows for actions in three places:

- entry actions (action executes on entry to a state)
- exit actions (action executes on exit from a state)
- transition actions (action executes on the transition)
- internal do actions (action executes continuously while in an action) [removed in 1.1]

Having four such types of action does not increase the 'power' of the notation. Consider, for example, a programming language that has a `while` construct and an `until` construct (equivalent to `while not( ... )`). The programming language is not more powerful for the addition of the new construct, it merely has a shorthand–which isn't necessarily clearer.

In the area of methods, however, more is often seen as better. There have been many method surveys (often notation surveys) that compare methods by counting 'meatballs', as shown in a table with methods along one side, and features along another. When a method has a feature, a meatball ('•') is placed at the intersection. Evaluating methods is then a simple matter of counting meatballs.

In some measure, the degree of notational sugar is a matter of taste, and there is certainly no requirement that you subset the notation. However, to speed time-to-market, there are several reasons why you may choose to subset it:

- It takes longer to learn four types of action than one, two, or three—multiplied then by the number of people involved, including domain experts and users.
- Because there are necessarily interactions between the four types of actions, readers of the model may not have the same execution semantics in mind as the writers.

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

- It sets up situations where developers can waste time arguing over matters of taste rather than matters of substance.

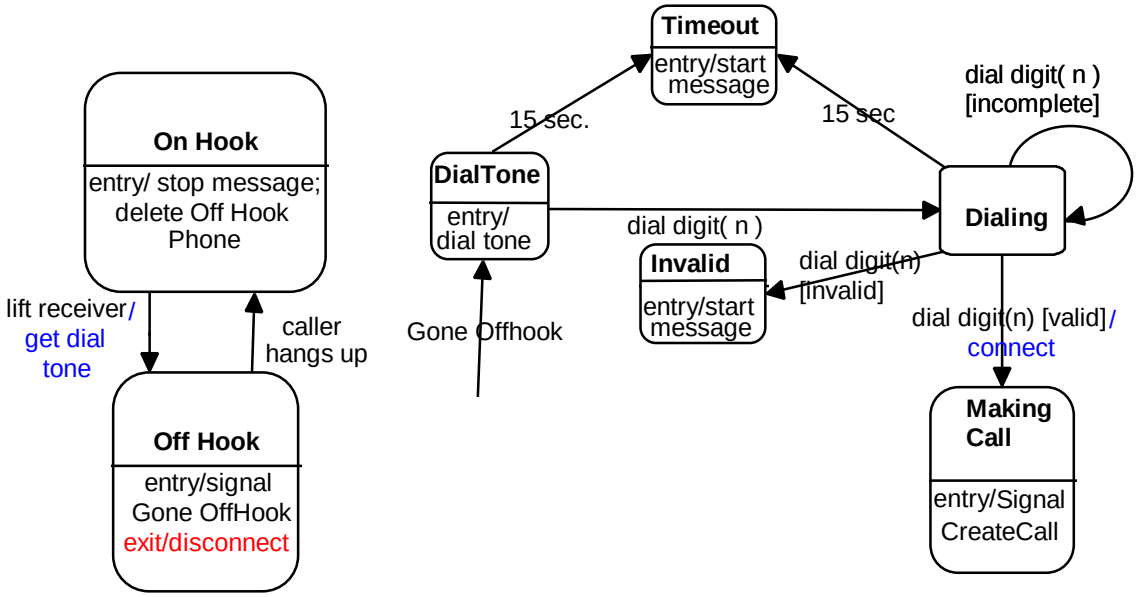


Figure 6: A comparable model without internal do actions.
Italics indicate changed elements of the model.

In particular, we recommend that you do not use internal do actions because this type of action has a special problem: it can obscure the nature of certain external processes. In the example model, it is not clear what the authors truly intended: Is the playing of a message truly continuous, or does it need to be started and left alone? What needs to be done to turn it off? There can also be misinterpretations of the semantics when a continuous action is interrupted. The message may turn itself off implicitly when it loses the channel, or it may require another action to terminate it. In the latter case, the model will contain actions to start the message, and in another state, to turn off the message, as shown in Figure 6.

In some cases, however, the action is truly continuous. Because computer systems do not act in a continuous manner, you can simulate the effect by periodically executing an action after an appropriate time. Model this using a transition to the same state with the expiration of a timer as the event.

A model without internal do actions may have a few extra transitions for the periodic execution of the action, and possibly a doubling of the processes that start and stop the "continuous" external process. Nonetheless, such a model will speed time-to-market through better understanding and less arguing.

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

There is no need to have actions on both states (either entry or exit), and on transitions. To subset the notation further, choose first whether to have actions on transitions or states; and, if you choose to put actions on states, choose between placing the action on entry or exit.

We prefer the simplest model in which we associate actions with states and execute them on entry–the Moore model [2]. This preference is largely a matter of taste, but we find that this approach encourages modeling the *lifecycles* of objects, rather than considering the events that cause responses.

We show the resulting models in Figure 7.

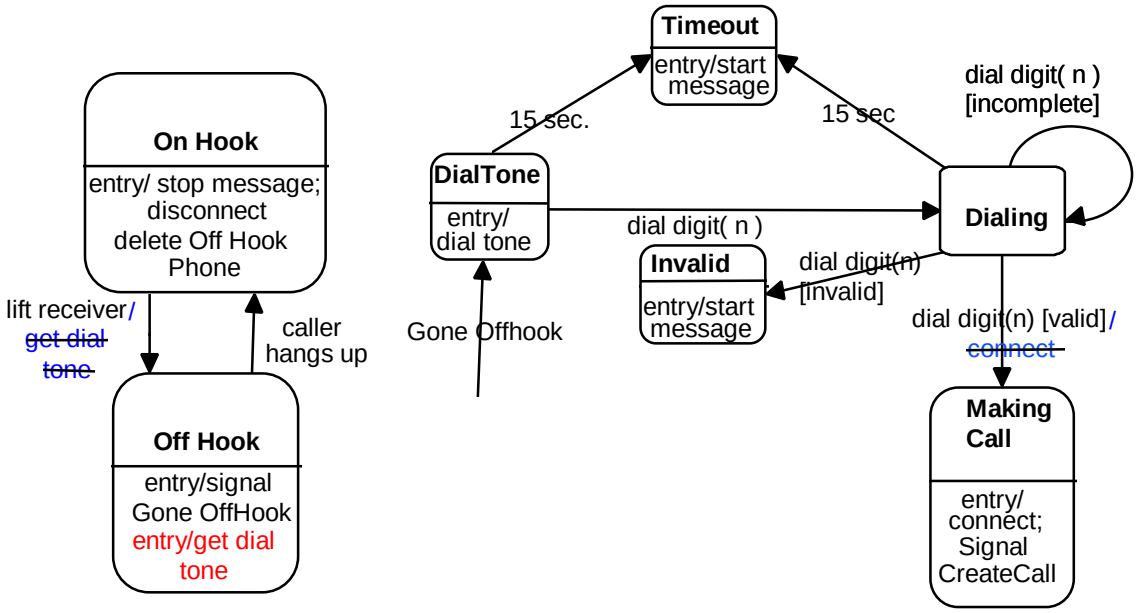


Figure 7: A comparable model without internal do, exit or transition actions.
Crossed out words represent elements removed as a result of changing the model.

6. Summary

The systems we build today are complex and they exhibit complex behavior. This does not mean, however, that the model of that behavior need be complex. In this paper, we have outlined four principles to simplify the models:

- Verifiable progress
- One fact in one place
- Accessible organization
- Simple but expressive notation

© Copyright 1997 by Project Technology, Inc.
All rights reserved.

All but the last focus on the *usage* of a notation–they are techniques for the management of complexity. The last suggests simply that a notation with a minimum of syntactic sugar is easier to learn and to use. The combination of these techniques and concepts will significantly reduce the time-to-market for the software in your project.

7. References

[1] UML Notation Guide, Version 1.1

[2] E. F. Moore "Gedanken-Experiments on Sequential Machines". In *Automata Studies*, Princeton University Press, Princeton N.J., 1956

© Copyright 1997 by Project Technology, Inc.
All rights reserved.