TEACHING SOFTWARE MODELING AND DESIGN BASED ON THE SCIENCE OF DESIGN AND SCIENCE OF LEARNING

Sukhamay Kundu
Computer Science Dept., Louisiana State University
Baton Rouge, LA 70803, USA
(kundu@csc.lsu.edu)

Abstract

Teaching software modeling and software design presents a different and difficult set of problems than teaching some of the other aspects of software engineering such as testing and requirements. As we point out this is partly due to the inherent complexity of the concepts involved in software modeling and design that requires a different approach in teaching them. The science of software modeling and design is also not quite as fully developed and mature although some major progress have been made recently. The lack of a good collection of practical and yet small enough examples of modeling and design problems for classroom use for illustrating both the science part of modeling and design principles and the engineering applications of those principles makes the teaching of these principles a significant challenge. We present a stepwise refinement approach for creating finite-state models that is better suited for classroom teaching.

1. Software Engineering Education

A good Software Engineering training involves several key components. On the one hand, we have team-work, software process (planning and management), and software evaluation (user-interface and performance); on the other hand, we have software requirements formulation, software modeling, software architecture, software design, software development and integration, and software (functional) testing. We focus here on the challenges faced in teaching software modeling and design.

2. Teaching Software Modeling and Design

We can view the finite-state behavior modeling of a software as building a "theory" of how the software should appear to the user whereas the subsequent steps of software design and development may be viewed as the two "engineering" steps to actually create a software that behaves accordingly (i.e., driven by that theory). We view the class-hierarchy structure of a software, which shows the organization of the data items and the operations (functions) in the software, as its design part.

A systematic method for creating an optimal class-hierarchy from the finite-state model and the use-relationship between the data items and the functions is given in [3]. A semi-automatic tool to facilitate the creation of an optimal class-hierarchy from an use-relationship is described in [4]. In [5], we showed that the finite-state model can also be used to group the functions of a software into components to create an architecture of the software before doing the class-hierarchy design. The success of both of these steps depends on building a correct finite-state model of the software. Although there is a systematic method for building a finite-state model from a given (finite) list of valid action (event) sequences and a (finite) list of invalid sequences [1], this is not suitable for practical applications such as modeling a software because there is no way of knowing beforehand how many valid and invalid action sequences would be needed. Thus, the approach in [1] is suitable for building an approximate model, which can be heavily influenced by the list of valid and invalid sequences chosen.

We remark that, in principle, one should be able to analyze the requirements of a software to determine the valid and invalid action sequences but this can be very tedious for a large software. If the requirements do not support such an analysis then this means requirements are not specified properly and we need to redo them. Note that we cannot expect the requirements to give us a regular-expression [6] for a complete description of all valid action sequences, and thus choosing a set of valid and invalid sequences becomes essentially an engineering step. For many applications, an extension of an invalid sequence is also invalid (i.e., once an error has occurred, it cannot be corrected by additional actions) and sometimes this can be useful in building a finite-state model. Our stepwise refinement approach is a practical method for building a finite-state model that alleviates the problems in [1] and that is suitable for classroom teaching.

2.1. Role of hierarchical information structure

Learning involves organizing the facts (concepts), their relationships, and the rules (methods) for applying them in a way that makes the recall (search) from the long-term memory during problem-solving easier [2]. In-depth learning means understanding not only more facts and their relationships but also the constraints associated with

each rule that determine when it is applicable. If the stored information in long-term memory is not easily searchable, then in a problem-solving session a student may not only fail to recall the proper information but also recall some improper information and unsuccessfully try to apply it.

The hierarchical information organization allow easier recall of relevant facts and rules during problem-solving. Thus, teachers need to structure the information presented to students in a hierarchical form whenever possible so that the students can organize (assimilate) the information more easily.

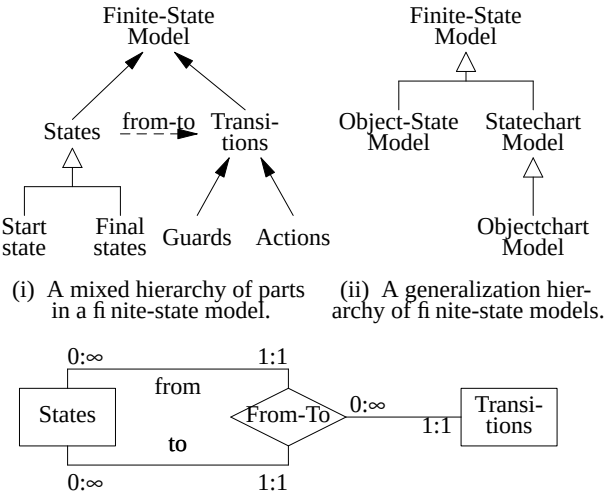
2.2. Generalization vs. compositional hierarchies

The generalization and compositional hierarchies are the two main types of information hierarchies. We need different approaches to build them, and thus we require different approaches to deliver the information in these hierarchies to the students for their assimilation. They also require different approaches for the recall and use during a problem-solving session.

In a generalization hierarchy, each item at level $L+1$ is built by addition of a small "chunk" of new information to an item at level L. This incremental building of complex information structure is key to its success in teaching and learning. We can traverse a generalization hierarchy in a top-down fashion by any combination of depth-first and breadth-first methods as needed to fit the different learning styles [7].

For a compositional hierarchy, a breadth-first scheme in assimilating (and teaching) the information is more suitable. Here, a top-down approach applies to visual (global) learner and bottom-up approach to sequential (analytical) learner [7]. Note that the conceptual or logical design of a compositional hierarchy is done more easily in a top-down approach although the actual physical construction of the hierarchy may require a bottom-up approach.

Indeed, there are cases where we need to combine the two types of hierarchies. The structure of a finite-state model falls in this category; see Fig. 1(i). Here, the three-way "from-to" relationship between transitions and states (which have two distinct roles "from" and "to") is quite complex and involves a number of cardinality constraints as indicated in Fig. 1(iii). For example, the constraint "1:1" next to Transitions indicates that a given transition has a unique pair of from-state and to-state. The constraint "0:∞" at the other end of the link from Transitions indicates that for a given pair of from-state and to-state there can be any number of transitions; the guards in transitions distinguish among these transitions. There are still other constraints such as "only one start-state", "one or more final-states", "every state is reachable from the start-state", etc; the third constraint helps to keep the number of states small. These constraints are not captured in Fig. 1(iii). There are well-known algorithms [6] to simplify a finite-state model by deleting unreachable states and minimizing the number of states by merging equivalent states. Fig. 1(ii) shows a generalization hierarchy of different types of finite-state models, which does not include models involving time-constraints (timed-automata) and parallel operations (petri-nets).



(i) A mixed hierarchy of parts in a finite-state model.

(ii) A generalization hierarchy of finite-state models.

(iii) An entity-relational model for "from-to" relation.

Figure 1. Two hierarchies of concepts related to finite-state models; the simple arrow '→' means "parts-of".

3. Difficulty in Teaching Finite-State Models

Unlike a generalization hierarchy, a compositional hierarchy of concept is more difficult to teach because it does not lend itself to incremental learning as easily. The full meaning of such a concept does not become clear until all the components of the hierarchy are understood. Recognizing misconceptions and correcting them are also more difficult for compositional concepts. Fortunately, finite-state models have simple visual descriptions in the form of state-diagrams, which help teaching this concept.

Determining a finite-state model for a given problem-statement (say, from the requirements description) is typically a very difficult task. We cannot determine the states one by one first and then determine the transitions one by one; the incremental discovery approach is not useful here. A good way to build a finite-state model is to create the behavior-tree of valid action-sequences (which is typically infinite, and thus we can begin by writing the tree upto some depth) and then try to merge nodes of the tree by arguing that the subtree at those nodes would be identical (if they were fully developed). As we merge more and more nodes, it becomes useful to create abstract characterization (description) of the nodes.

A trial-and-error approach to build the states and the transitions of a finite-state model is to be avoided because of the difficulty in correcting errors. An error can be due to the states and also due to the transitions. An error in transitions (or their guards) are comparatively easier to correct if the meaning of the states are clear. Note that a change in one transition may affect many valid and invalid action-sequences, and thus a change should be done with a full cognizance of which new sequences become valid and which previously valid sequences become invalid, with attention to the smallest sequences first. The situation is very different in correcting an error in states, which inherently requires adjustment of several transitions.

One good way to test the correctness of a model is to argue why deleting or adding transitions and merging of states would not work. Deletion of a state can be thought as a special case of deleting all transitions to that state to make that state unreachable from other states.

3.1. A small and yet challenging modeling problem

Consider a door of a room with a two-sided lock. We assume that initially the door is closed from outside and unlocked from both sides. The door can be locked from outside only if it is closed from outside, i.e., the person operating on the door is outside the room; likewise, the door can be locked from inside if it is closed from inside, i.e., the person is inside the room. We assume that the door can be unlocked only from the side it is locked. (For simplicity, assume that there is only one person who operates on the door; the person himself is not, however, involved in our finite-state model of the door-operations.) It follows that the door can be closed (and hence locked) only from one side at a time. Finally, the door can be opened from a given side only if it is closed but unlocked from that side. Once the door is open, it can be closed from either side; imagine the person operating on the door can move in and out of the room when the door is open.

Fig. 2 shows the correct finite-state model for the door, where we use distinct names for the outside and the inside operations. One reason for this is that they may involve different details of pushing/sliding the door in different ways. Also, the operations lockFrOut and unlockFrOut may involve a parameter for the key of the lock that must match with a predefined constant in order for these operations to be successful whereas the operations lockFrIn and unlockFrIn may involve a parameter that indicates, say, push button or remote control operation, without the need for key matching. The distinct names makes it quite easy to build the model in Fig. 2. The start-state is indicated by the bold circle; there are no final-states here in that the door-operations continue forever (or, alternatively, each state can be taken as a final-state). In state σ_2, the person may be inside or outside the room.

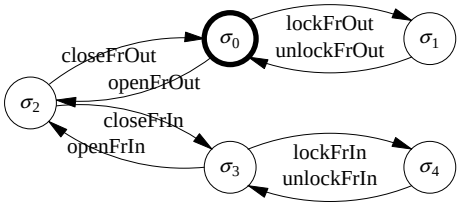


Figure 2. The finite-state model for the door with two-sided lock, with distinct names for all operations.

Now, assume that we use the same name for the inside and outside forms of the operations open and close. This is justified if we assume that a robot is operating on the door and at the center of the door (on each side) there is information that the robot can read regarding where the door-knob is located, which way it should be turned to open, and whether the door should be pushed or pulled for opening. This means a single open-function suffice for both forms of open, and similarly for the close-function. Fig. 3 shows the new finite-state model, which is considerably more complex. This model is easily obtained from Fig. 2 by replacing the names openFrOut and openFrIn by open and similarly for closeFrOut and closeFrIn, and then converting the resulting non-deterministic model (due to the close operation at state σ_2) to the deterministic form. However, it is not easy to build the model in Fig. 3 directly from the problem statement, and nor is it easy to imagine the original version of the problem as an intermediate step in building the model in Fig. 3.

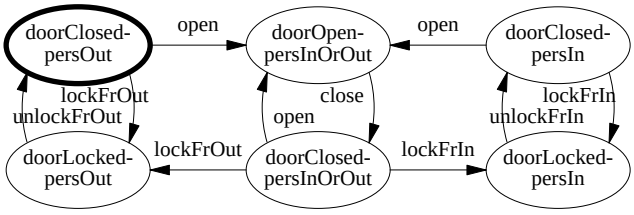


Figure 3. The finite-state model for the door when we use the same name for two forms of open/close.

It is instructive to look at the incorrect finite-state model in Fig. 4, which is conceived by several students in a software modeling course. The error in Fig. 4 is that lockFromIn (lockFromOut) can immediately follow unlockFromOut (unlockFromIn) although in reality they must be separated by an ⟨open, close⟩ operation sequence. In the next three subsections, we describe different approaches for building the model in Fig. 3.

3.2. Use of generalization in finite-state modeling

Sometimes modeling more than what is needed, i.e., a more special (detailed) case can be a useful tool in deriving the final model. This is the case for the finite-state model in Fig. 3 and we can derive it from the model in

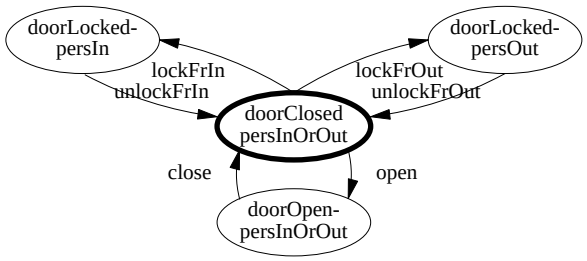


Figure 4. An incorrect finite-state model for the door.

Fig. 5, where we show the extra operations of the person (robot) going in and out of the room when the door is open. If we eliminate these extra operations by replacing them with λ-moves and convert the resulting non-deterministic model using the standard algorithms [6] into the deterministic form, we get the model in Fig. 3.

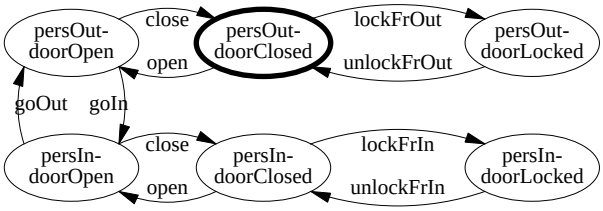
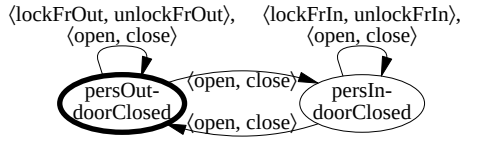


Figure 5. An extension of the model in Fig. 2 with goIn and goOut operations and using the same names "open" and "close" for their inside and outside forms.

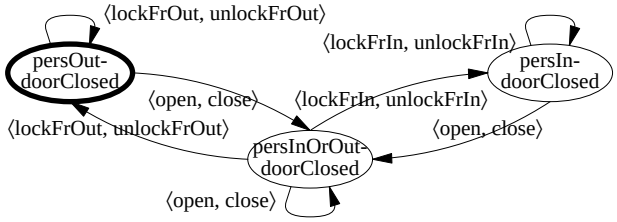
3.3. Finite-state modeling by stepwise refinement

Sometimes, it is easier to build a non-deterministic finite-state model first and then convert it to a deterministic model using the standard algorithms [6]. The deterministic model is needed for class-hierarchy design and architectural decomposition [3-5]. In addition, the modeling task can be simplified if we treat certain action-sequences, whose actions always occur consecutively in the order in those sequences, as a whole. For the door example, there are three such action-sequences: a = ⟨open, close⟩, b = ⟨lockFrIn, unlockFrIn⟩, and c = ⟨lockFrOut, unlockFrOut⟩; we can call each such action-sequence a compound-action. One can then unfold the complex actions by introducing additional new states on each transition involving a complex action. If a complex action is made of k simple actions, then we introduce $k-1$ new states in this process. The resulting model can be then minimized by the standard algorithms [6] to simplify it and obtain the final model.

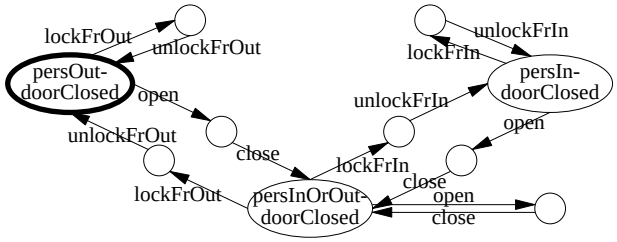
Fig. 6(i) shows a non-deterministic model for the door with two-sided lock considered in Fig. 3 using the compound-actions $\{a, b, c\}$. This model is easy to construct; here, the non-determinism occurs because of the action-sequence a = ⟨open, close⟩. Fig. 6(ii) shows the deterministic form and Fig. 6(iii) shows the result after unfolding it. The minimization of the model in Fig. 6(iii) gives the model in Fig. 3.



(i) A non-deterministic model of the door with two-sided lock using three compound-actions.



(ii) The deterministic model obtained from (i).



(iii) The unfolding of compound-actions in (ii) by adding new states.

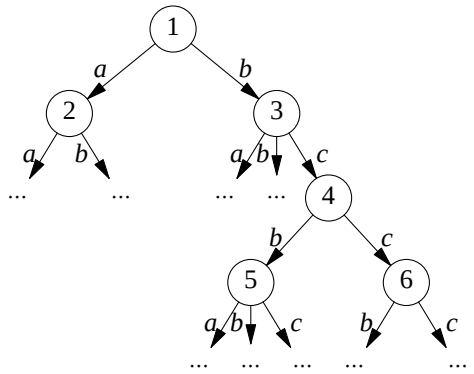
Figure 6. Illustration of the use of non-deterministic model and compound actions.

It is worth noting that if first build a model ignoring, say, the operations {lockFrIn and unlockFrIn}, then we only get the submodel on the states $\{\sigma_0, \sigma_1, \sigma_2\}$ in Fig. 2 with the names openFrOut replaced by open and closeFrOut by close. However, there is no easy way to enlarge it to get the model in Fig. 3.

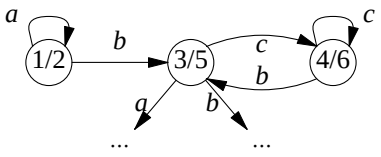
3.4. Finite-state modeling via machine learning

Fig. 7(i) shows a small part of the behavior tree, i.e., valid action-sequences for our door example up to the maximum length 4. Here, we are using the compound-actions a = ⟨lockFrOut, unlockFrOut>, b = ⟨open, close⟩, and c = ⟨lockFrIn, unlockFrIn> to keep the tree small. We briefly explain how we can determine the finite-state model in Fig. 6(ii) from the partial behavior-tree using a slight but useful variation of the machine-learning algorithm in [1]. We say two nodes in the behavior-tree are *equivalent* if the labeled subtrees at those nodes are identical based on the information in the partial behavior-tree. It is clear that no two nodes among {1, 3, 4} are equivalent to each other because they have different sets of child-

links. On the other hand, since we only know upto one level below node 2 and node 1 matches with node 2 upto one level, we conclude these nodes to be equivalent. (The algorithm thus uses an "optimistic" approach in deciding the equivalence of nodes.) For a similar reason, nodes 3 and 5 are equivalent and nodes 4 and 6 are equivalent.



(i) A partial behavior-tree for the model for the door with two-sided lock; see Fig. 3.



(ii) A partial finite-state model derived from (i).

Figure 7. Determination of the model in Fig. 3 from a partial behavior-tree.

We now build the finite-state model by merging equivalent nodes in Fig. 7(i). When we merge a node *x* with one of its ancestor node *y*, we remove all subtrees at *x* before the merging. Fig. 7(ii) show the result of merging the equivalent node pair {1, 2} (which makes the link (1, 2) into the loop at node 1), and merging the equivalent node pairs {3, 5} and {4, 6}. This leaves us the transitions for *a* and *b* in the merged state "3/5" undetermined. To complete the model, we develop the behavior-tree one or more levels for the *a*-child and *b*-child of node 3 in Fig. 7(i). (If we simply follow the algorithm in [1], then we would consider those two child nodes of 3 to be equivalent and merge them into a single node, giving a new state in Fig. 7(ii) and with no transition from that state.) It will be seen that those child nodes are equivalent, respectively, to nodes 1 and 3, and this would finally give us the model in Fig. 6(ii). The model in Fig. 3 is then obtained as before.

The above discussion clear shows that finite-state model building is a non-trivial task and requires a great-deal of practice and a combination of different techniques.

4. Need for Better Design Theory

Because there are only a few scientific principles or laws at present for designing a class-hierarchy, the training in software design relies more on engineering methods than a systematic algorithmic method. The existing rules-of-thumb for software design are often expressed informally and their application requires significant skills and creative thinking. We need better concepts in software design to alleviate the difficulties in teaching software design. One of the bottlenecks in software design is building a finite-state model, and we addressed that problem above.

5. Conclusion

We have shown that the basic task of building a finite state model for a software, which is the starting point for many high-level design tasks (e.g., an architecture for the software and a class-hierarchy for its object-oriented implementation), can be a daunting task even when there are just a few operations and a few constraints among them. We have argued that the inherent complexity of the information structure for a finite-state model prevents us from building a finite-state model in an incremental fashion. We have presented here a few techniques that can partially alleviate this difficulty and that can be easily taught in a class-room situation.

6. Reference

1. D. Angluin, Learning Regular Sets from Queries and Counterexamples, *Information and Computation*, 75(1987), pp. 87-106.
2. M.S. Donovan, J.D. Bransford, and J. Pellegrino (eds.), *How people learn*, National Research Council, 2000.
3. S. Kundu, Structuring software functional requirements for automated design and verification, *Proc. 31st Annual IEEE Intern. Computer Software and Applications Conference (COMPSAC-07)*, Beijing, 24-27 Jul, 2007 (nominated for best-appear award).
4. S. Kundu and G. Nigel, A formal approach to designing a class-subclass structure using a partial order on the functions, *The 29th Intern. Computer Software & Applications Conf., COMPSAC-05*, Edinburgh, Scotland, Jul 26-28, 2005, pp. 213-220.
5. S. Kundu, Orthogonal decomposition of finite-state behavior models: a basis for determining components in software architectures, submitted for publication.
6. H.R. Lewis and C.H. Papadimitriou, *Elements of the theory of computation*, Prentice-Hall, 1981.
7. L.C. Sarasin, *Learning style perspectives*, Atwood Publ, 1998.