On Abstraction in Finite State Machine Models

Michael Breen

12 September 2003 *(Draft)*

Abstract

Information hiding is a long established principle in software engineering. However, in the context of finite state machines a contrary principle of information exposure has been suggested. Using a simple example, we examine why a proven abstraction mechanism which works well in other areas may be less successful when applied to a finite state machine model.

After using Statecharts [3] to specify an aeronautical application, Leveson et al. [5, 6] report a difficulty that arose with hierarchical abstraction: Transitions involving superstates became difficult to understand when the substates they ultimately connected were shown separately on lower level diagrams. From this, they conclude that information hiding [7] impaired the specification's understandability. For requirements specifications, they recommend instead maximizing "information exposure."

The problem that we perceive and describe here relates neither generally to requirements specification nor specifically to Statecharts. It is a problem with abstraction peculiar to finite state machine (FSM) models.

To illustrate, we examine hierarchical abstraction as used in Structured Analysis, [2], with its equivalent in Real-Time Structured Analysis / Structured Design (RTSASD) [8, 4]. Figure 1 shows the general form of a data flow diagram (DFD) as used in Structured Analysis. A data transform in a DFD is labelled with an action describing what it does. For example, a transform called "Configure Alarms" might accept input data flows which indicate whether various faults are present and the severities assigned to these faults and produce an output flow indicating which alarms should be active. The "how" of a transform is then described at a lower level, perhaps using another DFD with more transforms.

Regardless of the efficacy or otherwise of the top-down approach itself, a data transform is generally suited to this kind of abstraction since it can be understood in terms of its function and its inputs and outputs. The relationship between data transforms and data flows is one of function and necessity: the function of the transform is to produce the output flows; to do this it needs the input flows.

As analogues to data transforms, control transforms in RTSASD accept control flow inputs and may be seen as transforming these into control flow outputs. Control flows carry instantaneous events and control transforms are ultimately specified

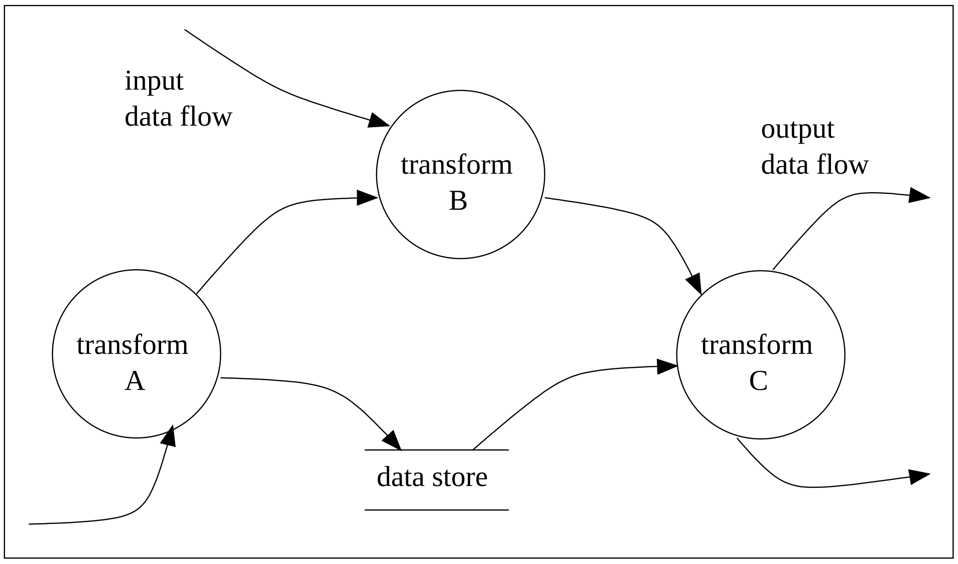

Figure 1: Data flow diagram.

as FSMs (flows which always have a defined value are also possible, as is stateless logic). The events cause the FSMs to change state and to emit further events (or signals) so that, in common with other approaches, the overall behaviour is modelled by a set of communicating FSMs.

Consider a model of part of the behaviour of a compact disc player: two control transforms, *Tray* and *Mode*, as shown in Figure 2 along with the corresponding state transition diagrams. As an example of how this operates, suppose that *Tray* is *Closed* and *Mode* is in state *Pause* and the open/close button is pressed. This causes *Tray* to change to *Opening* and emit the signal *s_stop* to which *Mode* instantaneously responds by changing to *Stop*. Note also that *Mode* is the selected mode and does not necessarily indicate the current activity: If the system is in state (*Stop*, *Open*) and the play event occurs, it changes to (*Play*, *Closing*) – meaning the CD will be played when the tray has finished closing.

For simplicity and to reduce clutter, most of the transitions in Figure 2 are unlabelled. Also, the FSMs have no outputs, that is, they don't actually do anything (such as activating a motor to close the tray) other than model a very simplified system state. Objections may be raised to the particular example, for example, it might be better in this case to replace *Tray* and *Mode* with a single FSM. However, in the general case, relatively independent aspects of behaviour will be modelled by separate FSMs which are synchronized as in this example to account for remaining dependencies.

Crucially, however, the control flows between the two control transforms are really a way of making the states and state transitions within a transform visible at its interface. For example, *s_close* in effect says that *Mode* is no longer in state *Stop* and so the tray needs to close. This is unlike a data transform which abstracts

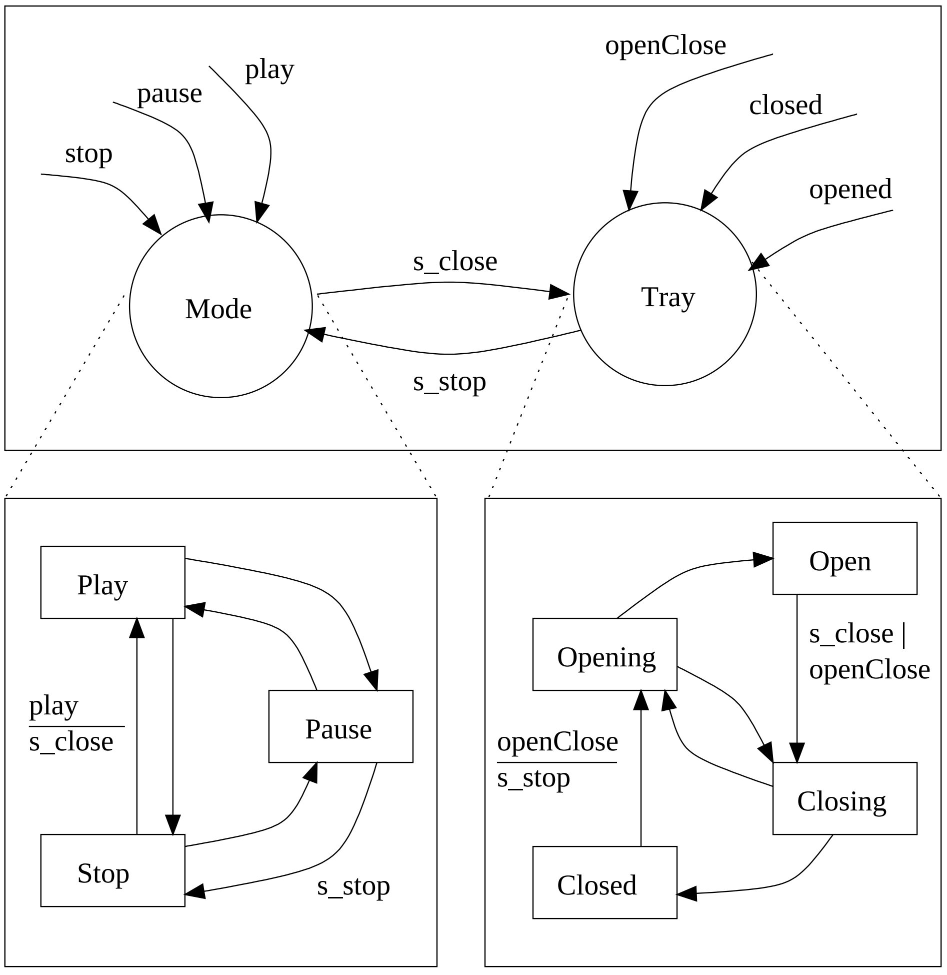

Figure 2: Control transforms and their state transition diagrams.

clearly some operation on data, or, for example, an object encapsulating the knowledge of whether some data are stored in an array or a linked list. Of what the control transform attempts or appears to hide, it is only the triggers and conditions on the transitions that can, in general, reliably remain hidden. Thus, though the diagram showing the control transforms and the interaction between them is superficially similar to a data flow diagram with data transforms, it differs in that it cannot fully be understood as a self-contained level of abstraction.

To make this clearer, imagine that, in order to model record functionality, a *Record* state is added to *Mode*. Also, *Tray* should not respond to presses of the open/close button when *Mode* is *Record*; since recording is a critical operation, it must be stopped explicitly by pressing the stop button. The simplest way to model this is simply to make the *Record* state visible to *Tray*, for example, by a flag (like a data store on a DFD) written to by *Mode* and read by *Tray*. An alternative might be to "filter" all the *openClose* events through *Mode* and simply have it relay them to *Tray* in every state except *Record*. However, even where the *Record* state is not directly made visible, the control flow which models the consequent dependency makes no sense unless you know there is such a state. In a sense, the FSM is the "what" as well as the "how" at the same level of abstraction; we cannot hide the latter without also hiding the former.

Further, the relationship between control transforms and the control flows between them is generally not one of function and necessity. Suppose the flag of the previous paragraph were omitted from the model – incorrectly allowing the tray to be opened even if *Mode* was *Record*. Unfortunately, this omission does not cause any obvious inconsistency or incompleteness: the *Tray* transform does not need the flag to operate nor is it the function of the *Mode* transform to write to it. In contrast to most similar cases in a data transform model, the only effect is a change in the behaviour in a particular combination of states. If it is inconceivable that this error could be overlooked it is only because the example used here is a trivial one.

Whatever the problems, abstraction still obviously serves an essential purpose in breaking up a model which could not legibly fit on one screen or one sheet of paper, thus making it physically manageable. However, one may find that the most effective way to understand and review such models is to print out all the sheets and lay them out on a large desk allowing one to skip around quickly between levels and FSMs and examine each part in the context of the others to which it relates. Thus, blind application of the so-called seven plus or minus two principle to show only a cognitively manageable chunk of data at once would be misguided. A large and complex FSM model may be better regarded as a map, which one breaks up only for reasons of practical necessity. The model of a digital watch in [3] provides a good example: By using a single large fold-out page for the statechart, the publishers allowed for a diagram which is clearly more comprehensible than the alternative of abstracting the contents of some of the higher level superstates on to separate diagrams.

Being able to see more of the model at once may also help to avoid errors similar to that involving the *Record* state in the example above. However, even if one can see the states of separate FSMs in reasonable proximity to each other, this in itself does not entirely solve the problem. To do this, one may systematically consider combinations of states in as efficient a manner as possible [1] In practice, this is not likely to be a major issue except in complex models where there are many possible dependencies between the different FSMs, such as may arise, for example, in certain kinds of interactive systems.

In conclusion, we have seen why FSM models may resist abstraction in a way that other kinds of model do not. While abstraction remains useful and is frequently necessary in such a context, one should be alert to its consequences in order to best decide whether it is necessary or beneficial in a particular case.

References

[1] M. Breen. (2003) Statestep specification technique: User guide. [Online]. Available: http://statestep.com

[2] T. DeMarco, *Structured Analysis and System Specification.* Englewood Cliffs, N.J.: Prentice Hall, 1979.

[3] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, July 1987.

[4] D. J. Hatley and I. A. Pirbhai, *Strategies for Real Time System Specification.* New York: Dorset House, 1988.

[5] N. G. Leveson, M. P. Heimdahl, H. Hildrith, and J. D. Reese, "Requirements specification for process-control systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 9, Sept. 1994.

[6] N. Leveson, M. Heimdahl, H. Hildreth, J. Reese, and R. Ortega, "Experiences using statecharts for a system requirements specification," in *Proc. Sixth International Workshop on Software Specification and Design*, 1991.

[7] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, Dec. 1972.

[8] P. T. Ward and S. J. Mellor, *Structured Development for Real-Time Systems.* Englewood Cliffs, New Jersey: Prentice Hall, 1985.