Statestep Specification Technique: User Guide

Version 2.0 / 2005-9-30

http://statestep.com

Contents

Acknowledgments

Modified compact disc recorder specification extracts appear by kind permission of Joop Kerssen, Philips Business Group Audio (Vienna). Thanks also to Marc Cools, Martine Looymans, Stefano Fontolan, John Mulcahy, James Curran, Donal Toomey, Rachel Moroney, Ben Toland.

Change History

1.1–1.6	Minor fixes and changes.
1.7	"Method in Practice" section re-organized and extended, particularly the subsection on "Choosing Variables"; other minor changes.
2.0	"About this Document" section added; "Checking for Correctness" appendix removed; '#' made the standard delimiter for line comments in formulas; several sections rewritten or improved.

1 Introduction

1.1 About this Document

This is primarily a guide to applying a particular specification technique – one that is supported by the Statestep software tool. You may also find it helpful if you are using Statestep for something other than specification.

This document is not a user manual for Statestep. It describes, in a software-independent manner, a simple notation and how to use it. The emphasis is on providing examples and practical guidance to help you to decide what to model, how to model it and how to deal with typical problems that may arise.

It should also give you an appreciation of the strengths and weaknesses of the approach and the kind of problems to which it is appropriate. It's important to understand that the notation is in no sense a "universal" language; rather, it is designed to be used for specific aspects of a system or model. This focus is an advantage: where it can be used, the method brings particular strengths to bear that would not be possible with a more general approach.

1.2 Why Use the Statestep Approach?

The Statestep specification technique may be described as a lightweight formal method of producing very high-quality specifications. Because the cost of specification errors rises dramatically the later in the development lifecycle they are discovered, this means the approach can dramatically

- improve product quality
- cut time-to-market and development costs

The chief advantages of the approach are:

1. **It is easy to understand.** In contrast to many "mathematical" formal methods, the notation is based on the use of straightforward tables. This makes the specification immediately comprehensible to all readers, including those from a non-computing background.

2. **It excels at elicitation.** As it makes all assumptions explicit, no unusual scenario is overlooked. Issues are identified and resolved upfront instead of being discovered later in the project lifecycle when they take longer to resolve and cause schedules to slip.

3. **It allows information to be recorded quickly.** This means the specifier can concentrate on gathering the facts about the required system behaviour rather than on premature design or structuring of the data.

4. **The specification is organized as a reference document.** The specification quickly answers any question about the system responses. If desired, it can also be mapped more-or-less directly to a set of test cases.

5. **Progress is measurable.** A relatively uniform desity of information means that the size of the specification provides direct feedback on how much has been done, while blank areas indicate which parts are yet to be considered.

6. **Specifications are precise and unambiguous.** The formality of the notation guards against misunderstandings and makes the specification a safe basis for agreement between developer and customer.

7. **Automated checks for correctness are possible.** Because the notation creates a model of the system (as a finite state machine), software can be used to check for specification consistency and completeness.

8. **Specifications are executable.** Code can be derived automatically from a specification, allowing (for example) prototype user interfaces to be produced for usability evaluation or the generation of a test oracle.

The main disadvantage is the physical size of specifications, which can be intimidating at first. However, this size is due to the use of tables, a page of which can be read much more quickly than (e.g.) a page of text. Further, specifications are designed to be used as reference documents, not to be read from cover to cover; they will normally be complemented by short, informal descriptions.

Using the approach also "front-loads" the work in a project and so the specification phase itself may actually take longer. However, this follows from all the work which should be done during specification actually being completed in that phase; the overall development time is reduced.

1.3 Domain of Applicability

The notation was originally devised to specify the external behaviour of an audio compact disc recorder (the source of the examples used in this document). The CDR may be characterized as a relatively small but surprisingly complex system. The complexity derives largely from the presence of a user interface which leads to potential feature interactions and a plethora of "interesting" scenarios (to take an example from the play functionality alone, "What happens if the machine is paused in the middle of a track, 'Repeat' is set to 'repeat track', a single digit button is pressed and, during the time allowed in case a second digit is entered to form a two-digit track number, the user presses the 'Next' button?")

The kind of system for which the notation is appropriate is most easily appreciated by learning the notation, the essentials of which are presented in just a few pages. In broad terms, it is suitable for systems or subsystems with complex finite-state behaviour, i.e., those likely to be modeled in a design as a number of interdependent or communicating state machines.

In terms of size, the notation cannot be used in its basic form to specify large systems. Extensions described in this document show how, as with other approaches, larger systems can be decomposed for the purposes of specification. Although this has been successfully done in practice, it does involve compromising on some of the advantages of the notation.

1.4 Comparison with Other Approaches

Two popular approaches, which contrast in different ways with Statestep tables, are scenarios and Statecharts. Scenario-based approaches share the merit of being easily understood. However they are unsystematic and so unusual scenarios can easily be missed. As a set of examples of typical system operation, they can never really be complete. Even to traverse all the (implicit) system states would require millions of scenarios for any realistic system. In contrast, a formal, model-based specification defines the system response to every possible sequence of events.

Note that this is not to suggest that scenarios and the like should be avoided. They can still be useful in providing an initial outline of system functionality (see Section 4.6). Such a text is likely also to serve as a good basis for a user manual for the end product.

Statecharts share the advantages of formal, model-based specification notations. The graphical notation is appealing as well as being powerful, efficient and subtle. However, in the context of specification (as opposed to design) this subtlety comes at a price:

1. Statecharts can be difficult to understand to those without a computing background for whom the concepts employed are unfamiliar.

2. The information is stored in a relatively indirect form and so the response of a complex system in a particular case is not readily apparent. Instead, one must simulate the Statecharts, discovering the effects of broadcast signals, keeping track of states and dependencies while moving up and down a hierarchy of states, perhaps in several different parts of the specification.

3. Negative cases are implicit. The absence of a response or a dependency in some part of the model does not reveal whether this is because the possibility has been considered and discounted or because it hasn't been thought about yet. Because they are always formally

complete, Statecharts cannot be checked for incompleteness and they do not lend themselves to systematic construction.

4. They do not afford systematic review. Simulations can in practice exercise only a small number of the possible behaviours and so the Statecharts, while defining how a complex system responds in every circumstance, do not necessarily model the required behaviour or the original intention of the modeler.

5. Because they are highly structured, the modeler must devote a lot of effort to the logical organization of the Statecharts. This distracts from the primary goal of simply recording the required behaviour. Also, because structuring decisions must be made as the data is gathered, re-organization (refactoring) may be required, with the danger of inadvertently changing correctly-modeled behaviour.

(Many of these disadvantages will of course be less evident in "toy" examples or even in larger, but comparatively simple, systems.) Points (1 – 4) also mean that Statecharts are relatively poor at eliciting unusual cases. Overall, the characteristics of Statecharts are much more those of a design notation than one suitable to be used for specification.

2 The Essentials

To introduce the notation, we use a simplified extract from the specification of an audio compact disc recorder. The principal sections of the specification are:

Variables to describe the state of the system

Constraints defining what states are possible

Events to which the system responds

Rules which describe the system's behaviour

As we will see later, there may be additional sections, e.g., to specify the contents of a front panel display in terms of the system state. Structuring is also possible, so that a system may be specified using more than one set of variables, events, etc.

2.1 Variables

This section lists the variables which, together, describe the state of the system at any given moment. Figure 1 shows some of this part of the CD recorder specification.

Figure 1: Variables for CD recorder.

Tray

open	The tray is fully open.
closed	The tray is fully closed.
closing	
opening	

Mode

The mode is the *desired* mode and does not necessarily indicate what's currently happening. For example, Mode may become play while the tray is open, in which case the CD will be played when the tray finishes closing.

play
pause
stop
record

2.2 Constraints

The variables describe the state of the system at any given moment; the constraints define which states are possible. For example, observe that if `Mode` is `play` the `Tray` must be either already `closed` (with the CD being played) or `closing` (in order to play it). This is an example of a constraint (or invariant).

Table 1 shows a constraint table which expresses this fact: From the row in which `play` is shaded, we read that whenever `Mode` is `play`, `Tray` must be either `closed` or `closing`. Similarly, the first row in the table indicates that whenever `Tray` is `open`, `Mode` can only be `stop`. The symbol '*' means "don't care." For example, if `Tray` is `closed` then the table tells us we can't infer anything about the value of `Mode`.

Table 1: Constraint table for CD recorder.

Tray	Mode	(other variables . . .)
open	stop	. . .
opening	stop	
closing	play pause stop	
closed	*	
*	stop	
closed closing	play	
closed closing	pause	
closed	record	
. . .		

The shading progress diagonally down the table and each variable value appears in a shaded cell exactly once. In general, then, to see the effect of a variable having a particular value on the other variables, simply read across the row in which that value is shaded. Every unshaded cell should be filled, either with one or more variable values or "don't care."

2.3 Events

This part of the specification lists the events to which the system responds. Figure 2 shows some of the events from the CD recorder specification.

An event is defined for anything that may cause the system to change state. For example, we know that an "end of track" event is necessary be-

Figure 2: Events for CD recorder.

KEY_OPEN_CLOSE	The "open/close" button is pressed on the front panel or on the remote control.
EV_TRAY_CLOSED	The (closing) CD tray reaches the fully closed position.
EV_END_TRACK	While playing a CD, the end of the current track (song) is reached.
EV_FAST_SEARCH_TIMEOUT	The "fast" button has been depressed continuously for two seconds.

cause we need to specify what the CD recorder does when this occurs (e.g., change `Mode` to `stop` if it is the last track on the disc). Note that the specification is not concerned with how the events will be detected in practice, only with defining them unambiguously.

2.4 Rules

The largest part of the specification comprises the transition rules which describe the behaviour of the system. Each rule specifies the response of the system to one of the events for some set of states. The rules are described in a transition table as illustrated by Table 2.

The first rule in this table (with identifier `r1_1`) specifies what happens when the "open/close" button is pressed and `Tray` is either `closed` or `closing` and `Mode` is either `stop`, `pause` or `play`. In all of these cases, `Tray` changes to `opening` and `Mode` changes to `stop`.

The second rule (`r1_2`) tells us that, if `Tray` is `open` or `opening`, `Mode` is `stop` and the "open/close" button is pressed, the system starts closing the tray. The '=' indicates that `Mode` is unchanged, i.e., it remains `stop`.

Note that the last rule shown applies regardless of the value of `Mode`. However, the constraint table tells us that `Mode` cannot actually be `record` when `Tray` is `closing` (only when it is `closed`). We could have listed the values `stop`, `pause` and `play` for `Mode` but it is easier simply to put in "don't care." It is not unusual to have rules which cover impossible states like this.

The transition table continues in this way, with rules covering every possible state for each event in turn. Even in those cases where there is no response, i.e., when the event is ignored, this is stated explicitly using a rule with "no change" symbols across the full width of the table.

Table 2: Transition table for CD recorder.

Tray	Mode	...
KEY_OPEN_CLOSE r1_1		
closed closing	stop pause play	
opening	stop	
KEY_OPEN_CLOSE r1_2		
open opening	stop	
closing	=	
...		
EV_TRAY_CLOSED r2_1		
closing	*	
closed	=	
...		

To conclude, some transition rule terminology. The *source states* of a rule are the states to which it applies. For example, the source states for rule `r2_1` in Table 2 include the state in which `Tray` is `closing` and `Mode` is `pause`. This rule defines a number of *transitions*, one of which maps this state to the *target state* in which `Tray` is `closed` and `Mode` is `pause`. In the table, the source states are given by the *source row*; the row following it is called the *target row*.

3 Extensions and Refinements

The basic notational forms and structure of the specification have now been introduced. However, in practice, even systems well-suited to being specified in this way will have aspects which do not fall neatly into this model. This section describes some refinements and additional elements which can be used in such cases.

3.1 Variable Value Hierarchies

When a variable has many possible values some table cells may contain long lists of values. However, it's often possible to summarise groups of similar values by arranging them hierarchically. For example, if our CD recorder allows for fast-forward during playback, the `play` value in `Mode` could be replaced by the values shown in Figure 3.

Figure 3: Grouping variable values

play-normal	Playback at normal speed.
play-ffwd-x2	Fast forward play (double speed).
play-ffwd-x8	Accelerated fast forward play (eight times normal speed).

With these values, `play-*` can be used as an alternative to listing all three play variations and `play-ffwd-*` can be used in place of the two fast forward values.

3.2 Constraint Formulas

Outside the constraint table, a constraint may be expressed as a formula, for example:

```
Mode = {pause | play} => Tray = {closed | closing}
```

which reads, "If `Mode` is `pause` or `play` then `Tray` must be either `closed` or `closing`."

Since anything looking like a mathematical equation is likely to be off-putting for some readers, it is a good idea to include an informal description of the constraint (like the one just given) along with the formula.

It's best to use the constraint table wherever possible. Not only is it easier to understand, the table also affords systematic identification of all constraints involving pairs of variables. Also, it is easier to use as a reference – it would take much longer to read through a list of formulas to find out what you want to know.

Unfortunately, it is not always possible to express a constraint in the table. For example, consider another CDR variable, `Drive`, with values which include:

stopped	The CD (if one is present) is not spinning.
tracking	The CD is spinning and the laser is following a groove (playing, recording or erasing).

Now, suppose we wish to say, "If `Mode` is `play` and `Tray` is `closed` then `Drive` must be `tracking`." This can be written as a formula:

```
Mode = {play} && Tray = {closed}
=> Drive = {tracking}
```

However, it should be clear that this constraint cannot be expressed in the constraint table (unless, e.g., we change the format somehow to allow `play` and `closed` to be shaded on the same row). If you want to formally describe a constraint like this then you have to use a formula.

The remaining operators used in constraint formulas are '`||`' (or) and '`!`' (not); parentheses may also be used. For example,

```
! Drive = {tracking} || Tray = {closed}
```

reads, "Either `Drive` is not `tracking` or `Tray` is `closed` (or both)," and

```
! (Mode = {record} && Tray = {open})
```

means it's not possible to be in record mode while the tray is open.

The order of precedence of the operators is given in Table 3. For those who require it, the full grammar can be found in Appendix A.

Table 3: Operator precedence (highest first).

`!`	NOT
`&&`	AND
`||`	OR
`=>`	implies

3.3 Event Contraints

The constraints described in the previous section were *global constraints* – constraints which always apply. It often happens that an event is possible only in certain states. To express this fact, we can use an *event constraint*.

For example, a "tray closed" event for the CD recorder can occur only in states where `Tray` is `closing`. This constraint is expressed as a formula, i.e.,

```
Tray = closing
```

and listed as an event constraint under the corresponding event. (Note: there is no constraint table for event constraints).

Even when a constraint is obvious, like this one, including it in your model can prevent spurious errors being reported. For example, software might otherwise warn you that that you've provided no rule that says what happens when a "tray closed" event occurs when the `Tray` is `opening`.

3.4 Dealing with Data Aspects

In most systems, there will be elements which are not suited to being modeled using simple state variables like `Mode` and `Tray`. These often have a numeric character and may be called *data aspects*. An example of this is the program (or "memory") function on a CD player or recorder which allows an arbitrary sequence of tracks to be stored and played back.

While data aspects cannot be incorporated directly into the finite state-based model, it is sometimes useful to abstract equivalence classes from them. Figure 4 shows what can be done for the CDR program.

Figure 4: Program Variable for CD recorder.

Program

This models the principal cases of interest relating to the sequence of tracks stored for playback.

empty	The program memory is empty.
entered	A program exists, i.e., at least one track has been stored but there is still space in memory for additional tracks.
maximum	The maximum number of tracks has been stored in the program.

The coarser aspects of the program functionality can now be included in the formal model (e.g., that it is not possible to add another track when the program is already full). Moreover, having this variable represented in the transition table is useful for elicitation since it forces us to consider it as each rule is defined (e.g., "Should the program be cleared when the system enters record mode?").

The finer details may then be given in comments accompanying the rules. For example, consider Table 4 which describes what happens in two cases where the user attempts to add a track to the program by pressing a digit button in the new mode, `program`. (To simplify the example, we

assume that a CD can have no more than nine tracks, i.e., that a single digit specifies a track. In reality, an audio CD can have up to 99 tracks.)

Table 4: Rules for program functionality.

Tray	Mode	...	Program	...
KEY_DIGIT r9_4				
closed	program		empty entered	
=	=		entered maximum	
The digit pressed is the number of a track which exists on the CD; this track is added to the end of the program.				
KEY_DIGIT r9_5				
closed	program		empty entered	
=	=		=	
The digit pressed is not the number of a track on the CD.				

Table 4 illustrates the use of abstraction not only with respect to the program but also an event: `KEY_DIGIT` is used to model the pressing of any of the ten digit buttons. The numbers of the tracks on the inserted CD are also omitted from the formal model. A natural language description provides the missing information.

Of course, as an alternative to a simple comments, you might choose to refer to diagrams elsewhere or use any other suitable means to deal with the data aspects. For example, statements in a programming language could be added to a rule to manipulate data stores; if code was to be generated from the model later then these statements could automatically be incorporated at the appropriate point. Formal methods experts might even decide to use a mathematical notation such as 'Z' to add precise descriptions of operations on data. It all depends on how readable or how precise you need your specification to be.

Note that this kind of abstraction results in *non-determinism*, that is, uncertainty: In rule `r9_4`, for example, it cannot be determined from the model alone whether `Program` changes to `entered` or `maximum`. Naturally, we expect that `Program` changes to `maximum` only if the new track takes the last available slot in memory. However, even if we add a comment or a more formal description to the rule, this fact will not be part of the Statestep model. Further, when the notes added to the rules above are disregarded, the two rules seem to apply in exactly the same set of cases. Consequently, automated checks on the model may produce nuisance warning messages.

3.5 Repetition in Rules

In certain situations, the rules in a transition table may become undesirably repetitive. As an example of one kind of situation in which this may arise, consider another CD recorder variable, for (appropriately) a 'repeat' function, shown in Figure 5.

Figure 5: Definition of CDR variable Repeat.

Repeat	
no_repeat	The repeat function is disabled.
repeat-one	When a track is finished playing, it is played again. This continues indefinitely.
repeat-all	All the tracks on the disc or in the program (if there is one) are played repeatedly.

Now imagine that we wish to specify that the repeat setting toggles on pressing the "repeat" button, but only when either

- the tray is not closed, or
- the CDR is in stop mode

Table 5 shows the most straightforward way of describing the desired behaviour. Note that there are two cases in which `Repeat` toggles; also, to describe the toggling between the three values separately would require three rules. However, the total number of rules used in Table 5 is the *product* of these two numbers, i.e., six. For similar cases, even if they are rare, the result of this effect could be much worse.

The simplest way to deal with this is to use abstraction as in Section 3.4, i.e., to move the description of the toggling behaviour out of the formal model into a comment. This approach is shown in Table 6. In this case at least, it is probably the best strategy. Another alternative is to express the rules using formulas (to be covered in Section 3.6).

Another possible cause of repetition is the inclusion in the transition table of a variable the value of which depends only on the current values of other variables. For example, the contents of the front panel display of the CD recorder are determined by whether the tray is closing (display a message "closing"), closed in 'stop' mode (display the number of tracks on the CD) and so on – see Section 3.7.

Table 5: A repetitive way of specifying “repeat” toggling.

Tray	Mode	...	Repeat	...
KEY_REPEAT r11_2				
open opening closing	*		no_repeat	
=	=		repeat-one	
KEY_REPEAT r11_3				
open opening closing	*		repeat-one	
=	=		repeat-all	
KEY_REPEAT r11_4				
open opening closing	*		repeat-all	
=	=		no_repeat	
KEY_REPEAT r11_5				
closed	stop		no_repeat	
=	=		repeat-one	
KEY_REPEAT r11_6				
closed	stop		repeat-one	
=	=		repeat-all	
KEY_REPEAT r11_7				
closed	stop		repeat-all	
=	=		no_repeat	

Table 6: Removing repetition by abstraction.

Tray	Mode	...	Repeat	...
KEY_REPEAT r11_2				
open opening closing	*		*	
=	=		*	
Repeat toggles in the following sequence: no_repeat, repeat-one, repeat-all, no_repeat, ...				
KEY_REPEAT r11_3				
closed	stop		*	
=	=		*	
(See r11_2 for the repeat toggling sequence.)				

3.6 Using a Formula to Define a Rule

Rule formulas are not essential to the notation and can be subtle. You may wish to skip this section or leave it to a later reading.

As with constraints, it is possible to specify a rule using a formula – or a combination of table and formula. Though it's never necessary to use a formula in a rule, it may sometimes be the only way to avoid having to define lots of repetitive tabular rules if you want to keep to a formal description of behaviour within the Statestep model.

The syntax is similar to that used for constraints, but with one extra operator, '`->`', to specify the new value of a variable. To illustrate, consider again rule `r1_2` from earlier, repeated in Table 7. As a formula, this rule could be written:

```
Tray = {open | opening} && Mode = {stop}
&& Tray -> {closing} && Mode -> {=}
```

The symbol '`->`' is read "becomes" or "changes to".

A rule formula says what is true before and after the rule is applied; if there is no way to make the formula true in a particular case then the rule does not apply. For example, the formula above can be satisfied if `Tray` is either `open` or `opening` and `Mode` is `stop` – we need only to make `closing` the new value of `Tray` and make the new value of `Mode` the same as its current value, that is, `stop`.

A mistake in a rule formula may mean that the rule never applies. Suppose the above formula were changed to

```
Tray = {open | opening} && Mode = {stop}
&& Tray -> {closing} && Tray -> {=}
```

Table 7: "Closing" rule for CD recorder.

Tray	Mode	...
KEY_OPEN_CLOSE r1_2		
open opening	stop	
closing	=	

This is a valid formula (even though it doesn't specify a new value for `Mode`). However, it requires that the new value of `Tray` is `closing` and also that the new value of `Tray` is the same as its current value. Thus, it requires that the current value of `Tray` is `closing`. But it also applies only when the current value of `Tray` is either `open` or `opening`. This contradiction means there is no way to satisfy the formula and the rule might as well not exist.

The implication ('`=>`') operator is best avoided in rule formulas as it almost always indicates another kind of mistake. Suppose we replace one of the logical ANDs in the first formula with an implication operator:

```
Tray = {open | opening} && Mode = {stop}
=> Tray -> {closing} && Mode -> {=}
```

Now, recalling the basic rules of logic, a formula in the form "`X => Y`" is equivalent to "`(X && Y) || !X`". This means that *the formula above always applies*, regardless of the values of `Tray` and `Mode`: it does say what the new values of the variables are if their current values are `open` or `opening` and `stop`, respectively; however, it doesn't require that these be the current values. For example, if `Mode` is `play` then the formula can still be satisfied, it's just that the new values of the variables are unspecified in that case, that is, the variables are allowed to change to any value. The formula is valid but it's almost certainly not what was intended.

The '`->`' operator is flexible. A concise alternative to the original formula above is

```
Tray = {open | opening} -> {closing}
&& Mode = {stop} -> {=}
```

In other words,

```
Tray = {open | opening} -> {closing}
```

and

```
Tray = {open | opening} && Tray -> {closing}
```

are equivalent.

It's also possible to string a sequence of '->'s together to more concisely describe toggling behaviour. Thus,

```
Repeat = {no_repeat} -> {repeat-one}
-> {repeat-all} -> {no_repeat}
```

means the same as

```
Repeat = {no_repeat} -> {repeat-one}
|| Repeat = {repeat-one} -> {repeat-all}
|| Repeat = {repeat-all} -> {no_repeat}
```

To illustrate how a formula can be useful, consider again the example in Section 3.5. Using a formula to define the toggling, the six rules in Table 5 can be replaced by the two rules in Table 8.

Table 8: Using a formula to specify toggling.

Tray	Mode	...	Repeat	...
KEY_REPEAT r11_2				
open opening closing	*		*	
=	=		*	
Repeat = {no_repeat} –> {repeat-one} –> {repeat-all} –> {no_repeat}				
KEY_REPEAT r11_3				
closed	stop		*	
=	=		*	
Repeat = {no_repeat} –> {repeat-one} –> {repeat-all} –> {no_repeat}				

Note that "don't cares" are used in the table cells for the variable `Repeat`. However, the rules are still deterministic because the formula (unlike a textual comment) is part of the model and the rule is defined by composing (ANDing) the two parts, table and formula. So, for example, writing rule `r11_2` in Table 8 entirely as a formula gives us[1]

```
Tray = {open | opening | closing} -> {=}
&& Mode = {*} -> {=} && Repeat = {*} -> {*}
&& ( Repeat = {no_repeat} -> {repeat-one}
-> {repeat-all} -> {no_repeat} )
```

[1]The variables omitted from the table to simplify this example are, of course, also omitted from the formula.

or, equivalently,

```
Tray = {open | opening | closing} -> {=}
&& Mode -> {=}
&& ( Repeat = {no_repeat} -> {repeat-one}
-> {repeat-all} -> {no_repeat} )
```

We could go even further and use just one rule as shown in Table 9. However, this is probably not a good idea. One of the main advantages of the tabular format is that one can scan it quickly, taking in the information at a glance. In contrast, the formula in Table 9 has become quite difficult to parse. It will be off-putting to many readers and complex formulas are more likely to contain errors.

Table 9: A single rule for Repeat toggling.

Tray	Mode	...	Repeat	...
KEY_REPEAT r11_2				
*	*		*	
=	=		*	
(Tray = {open | opening | closing} || Tray = {closed} && Mode = {stop}) && (Repeat = {no_repeat} –> {repeat-one} –> {repeat-all} –> {no_repeat})				

It is best to complete all table cells whether or not the corresponding variables appear in the formula. As in the examples above, this can be done by entering "don't cares" or the values allowed in the formula, as appropriate. Following this convention means that empty cells can be taken as a sign that the rule hasn't been completed. For example, if a new variable is added to the model, the empty cells in the new table column will serve as a visual reminder that it hasn't yet been considered whether that variable really is a "don't care" for that particular rule.

3.7 Displays and Other State-Dependent Outputs

To specify an output which is determined by the values of some or all of the variables, a separate condition table is used. Table 10 illustrates the use of this format to specify the contents of the front panel display of the CD recorder. (Table 10 includes a new variable, `DisplayMode`, which is a common feature on CD players and recorders. Though we haven't seen it so far, it would of course have been included in the constraint and transition tables.)

Why not specify the display within the transition table? This could be done using a variable `Display` with values corresponding to the various

Table 10: Condition table for CDR display.

Tray	Mode	...	DisplayMode	...
open opening	*		*	
"OPEN"				
closed	play pause		elapsed	
The track number and the elapsed track time, e.g. "3 1:10".				
closed	play pause		remaining	
The track number and the remaining track time, e.g. "3 2:07".				

possibilities (e.g. `open_msg` for "OPEN", etc.) It would then be possible to see not only the effect on the system state of an event but also how the display changes all in one place, i.e. in each rule in the transition table.

Unfortunately, this would introduce a great deal of redundancy into the transition table and make it much larger than it needs to be. For example, Rule `r2_1` in Table 2 would have to be replaced with two rules as in Table 11. Many other rules would also have to be split in a similar way. In effect, the information given once in the condition table would end up being repeated in many places throughout the transition table.

Table 11: Including a display variable in the rules.

Tray	Mode	...	Display	...
EV_TRAY_CLOSED r2_1a				
closing	play pause		*	
closed	=		track_time	
EV_TRAY_CLOSED r2_1b				
closing	stop		*	
closed	=		num_tracks	

3.8 Structuring Larger Specifications

This is an outline that is most likely to be useful to experienced modelers. A proper tutorial with examples would require a much longer treatment.

One of the merits of the notation is that the response of the whole system for any combination of state and event can be seen in one place. However, when the number of state variables becomes greater than about 18, difficulties arise. For example, a table with more than this number of columns rendered in a readable font size will probably be too wide to fit all at once on a computer display or to be printed on standard A4 paper (even in landscape orientation). If the notation is used in such cases then it is necessary to structure the specification in some way, i.e., to break it down into cohesive parts. Each of the parts is then described by separate tables (or perhaps some parts will be more suited to some other notation).

A broad outline of the approach taken to specify a dual-tray CD recorder illustrates this. The additional tray allowed the contents of the CD in one tray to be recorded directly onto the other. Since only one of the trays included record functionality, the behaviour of the two differed somewhat, but much was common to both. In the specification, one transition table described the common responses and two more tables were used for the behaviour unique to each tray.

Though this division was a good one, the two trays were not completely independent. For example, to ensure synchronization, it was possible to start recording one CD and start playing the other in a single action. Also, the response of one tray to an external event sometimes depended on the state of the other.

This interdependency was modeled in two ways. First, signals (i.e., internally-generated events or messages) were defined. Any of the rules for one subsystem (tray) could specify, as part of a comment accompanying the rule, that a signal be sent to the other. For example, the record tray could send a signal to the playback tray requesting that it start playing its CD. The response of the receiving subsystem was specified in the same way as its response to an external event, i.e., there was a set of rules in its transition table for each of the received signals.

Second, certain aspects of the system state were represented in both transition tables. In other words, the same variable appeared in both – or, more generally, the value of a variable in one table could be derived from the value of one or more variables in the other.

The specifier is free to use whatever structuring primitives are useful. In doing so, however, it is important is to have a good intuitive feeling for the semantics of the notation and to respect them. For example, with only one transition table, the response of the whole system in a particular state is shown in a single rule in the table. By extension, with two tables, each one should show the response of the part of the system that it models. This would not be the case in a specification which allowed the following to happen: As a response to an external event, a rule in the transition table for one subsystem specifies that a signal is sent to another. A rule for the second subsystem, specifying the response to this signal, specifies that

another signal is in turn sent back to the first subsystem. Thus, to see the "final" state of the first subsystem, one would need to refer to more than one rule in its transition table. To use the notation in this way would be to turn it into little more than a programming language.

It is useful to regard internal events and other such primitives simply as labels giving directions on how to join up the tables into one big one.[2] Thus, if a rule for one CDR tray sends a signal `START_PLAYBACK` to the other, don't visualise a message with some existence of its own being sent inside the actual system. Instead, read it as, "To see the response of the other tray, refer to the section marked `START_PLAYBACK`."

This conception also provides the answer to a question that can arise about shared variables: "Suppose `A` sends a signal to `B` and the variable `X`, which appears in the transition tables for both subsystems, changes its value. In which transition table should it be shown changing state? I don't want it to change in `A` because what `B` does depends on the old value which will have been overwritten..."

Again, the mistake is in imagining some mechanics which don't really exist, i.e., that `A` does something first, tells `B` and then `B` does something else. There is no sequence. What happens in the rule in `A`'s transition table should be imagined as happening simultaneously in the rule for subsystem `B`. Therefore, show the variable changing value in both tables. Alternatively, if it's possible to know what the new value of the variable is in only one of the tables, show the new value there; in the other table put a "don't care" for the new value of the variable. (In the target row, "don't cares" may be read as "don't knows", i.e., "look somewhere else to find out").

Structure should be introduced into the specification only as needed. Wherever practical, defer structuring until the design stage. Signals and other primitives in the specification add to the concepts which must be understood by those reading the specification, not all of whom may have a computing background. After some experience with the tabular notation, even engineers who used the structuring described above regarded it as a kind of necessary evil, something they felt really belonged in the implementation; ideally, all the behaviour would be in one big table.

[2] If you are familiar with relational algebra, think of a natural join.

4 Method in Practice

4.1 Choosing Variables

Choosing variables can turn out to be a lot less straightforward in practice than it might appear from looking at ready-made examples. Generally, variables represent either (1) some aspect of the system state that is observable by the user or (2) something that is not directly observable but must be remembered by the system if it is to behave correctly.

An example of the first kind would be a variable named `Fan`, corresponding to a cooling fan in some system and having as possible values `on` and `off`. This is the easiest kind of variable to understand: one that maps directly to something physical that can be seen or heard in the real system. A rule in which the value of the variable changes shows the visible system response directly – in this case, as a fan switching on or off.

Now, suppose the fan should switch on when a certain temperature is reached but that it should not switch on unless an activation button has been pressed since the system last reset. Clearly, this means the system must remember whether the button has been pressed or not. To model this, we use another variable. In this case, the variable doesn't correspond to anything the user can see directly – you can't tell just by looking at the system whether the button was pressed or not. However, it is necessary if the system is to function correctly.

Assuming we use a simple boolean variable (with, e.g., `true` and `false` or values `yes` and `no`), how should this variable be named? One possibility is to use something like `ButtonWasPressed`. However, `FanEnabled` would probably be a better name: this gives some indication of the purpose of the variable. In fact, it would also be better to give the values more descriptive names, say `enabled` and `disabled`; this allows one to understand more quickly the information in a row of a table, without necessarily looking up to the column headings to see what a generic `true` or `false` relates to. However, for this example, we'll leave them at `true` and `false`.

Now that we have two related variables, another question arises: would it be better to replace these two variables with a single variable? Table 12 shows how this could be done: the possible combinations of values of the two variables `Fan` and `FanEnabled` are on the left, each mapped to the equivalent value of an alternative `Fan` variable on the right. (Note that the combined variable doesn't fall neatly into either of the categories mentioned above).

The main advantage of reducing the number of variables is that it reduces the number of columns needed in the tables. Often, this is a compromise: while the number of columns may reduce, you may find the number of rules needed to represent the same behaviour increases (we'll see an example of this presently). Obviously, only variables that are related in

Table 12: Replacing Fan and FanEnabled with a single variable.

Fan	FanEnabled	Fan
on	true	on
off	true	off-enabled
off	false	off-disabled

some way should be considered as candidates for being combined like this. Apart from this, one has to decide on a case-by-case basis whether it is a good idea or not. However, a reasonable rule of thumb is to check whether the total number of variable values increases or decreases. In Table 12, we see the total number of variable values needed with two variables is four (two each for `Fan` and `FanEnabled`) while using the alternative `Fan` variable means just three values have to be defined – suggesting this option might not be a bad one.

It's good to think about alternative ways to model the system state even where the number of columns is unchanged. Consider, for example, the variables `Tray` and `Mode` introduced in Section 2.1. This choice is a good one but it is not as obvious as it might appear. One of several plausible alternatives is to have a variable reflecting what the CDR is currently doing, say `Operation` and another, `Pending` to indicate what the system is to do when it's finished its current action. Thus, if the "play" button is pressed when the tray is open, `Operation` becomes `closing` and `Pending` changes to `play`. Table 13 shows the possible combinations of values of these variables along with the equivalent combinations of `Tray` and `Mode` values.

Table 13: Alternative to Tray and Mode variables.

Tray	Mode	Operation	Pending
opening	stop	opening	none
open	stop	open	none
closing	stop	closing	none
closing	play	closing	play
closing	pause	closing	pause
closed	stop	stop	none
closed	play	play	none
closed	pause	pause	none
closed	record	record	none

In fact, `Operation` and `Pending` are a poor choice. This could be ar-

gued on the grounds that no variable maps directly to the physical tray or the feeling that `Pending` is "not doing very much". Also, the number of variables values needed with the `Operation / Pending` variation (ten) is greater than that needed with `Tray` and `Mode`. A further criterion by which to compare the alternatives is the number of rules required to model the system behaviour in the transition table. In Table 2, the closing of the tray is modeled using only a single rule (`r2_1`). As the reader may verify, modeling (without formulas) the equivalent transitions using the variables `Operation` and `Pending` requires three rules.

Although it may be necessary to balance a variety of considerations, the most important thing is to aim for variables which can be easily understood and which are reasonably independent of each other. There is inevitably a certain amount of skill involved in choosing good variables to model a system state; and even with experience one can still find, with the benefit of hindsight, a better way to do it when the specification is half-finished. Unfortunately, refactoring variables can mean a lot of work changing rules, so it's best to try to develop an awareness of the possibilities and get it right at the start if possible – though not of course to the extent that one sinks into "analysis paralysis".

4.2 Abstraction and Deciding What to Model

It's important to remember that it's not necessarily a good idea to try to represent everything in the tabular model. Indeed, it's often impossible to do so anyway. We've already seen, in Section 3.4, how a variable with an enormous number of possible values (i.e., every possible sequence of tracks from a CD) may be abstracted to a finite state variable with just a few values, corresponding to the most important cases to consider.

In other cases, it might not be appropriate to model even an abstracted version of non-finite state data. Suppose, for example, a system has an "undo" feature. If this allows only the last command to be undone, it might be possible to model this behaviour completely in the tables, using a variable which has as its value the last event corresponding to a user action. If more commands can be undone then one might include in the model, e.g., a variable which indicates whether there any commands in the undo buffer.

However, either way, the question to be asked is: Is it useful to model "undo" within the tables? Among the main advantages of describing behaviour in the tables is that unusual cases are not overlooked; if adding a variable to help model undo functionality within the tables doesn't improve the chances of this, then it should be considered whether to try modeling the behaviour in the tables at all. Instead, one might describe the undo functionality somewhere else, in natural language or otherwise, giving details such as how big the undo buffer is, which commands can be undone,

etc.

We also want to model the state of the sytem only to the extent that it helps an external observer to understand the behaviour. When describing the user-visible behaviour of a laser printer, a "warming up" state helps to explain why there's a delay after switching on the printer before it's ready to print; however that doesn't mean the internal component that's warming up needs to be specified. Similarly, if a system goes through a series of self-tests which take some time to complete and during which it doesn't respond to any user action then there's probably no point in modeling a series of "testing X" states - a single "self-testing" state should be enough, with the system exiting that state on a "tests complete" event.

Of course, the "tests complete" event we just mentioned is somewhat artificial, but we are free to define it as a convenience for the sake of the model. In the real system, this probably isn't something that's detected as such, but rather corresponds to reaching the end of a sequential procedure; it is simply that the model requires that every change in state be associated with an event. In general, we're not concerned with how an event is detected, only that it can be detected. For example, we might use an event defined to occur ten minutes after the ambient temperature first exceeds a certain value, A, provided the temperature remains at or above that level throughout the ten-minute period. In the real system, this event might be detected by continually polling a sensor and keeping track of the times the temperature went above or below A, or in some other way – but that is of no concern in the specification.

4.3 Iterative Specification

A specification is not produced in the same orderly way in which it finally appears, with all the variables being defined first, then the constraints, and so on. In practice, some obvious events such as button presses may be defined first, then a couple of variables may be chosen to model the more significant aspects of the system state. After considering possible constraints, it is discovered during definition of the rules for one of the events that some additional state information is necessary. This is included, either by adding another variable or by adding a new value to an existing variable. The question arises as to what happens in one of the new states in response to one of the other events. Rather than completing the rules for the first event, the answer to this question is immediately recorded as a rule for the second event. The specification process thus proceeds in an iterative fashion and the required behaviour is recorded in the more-or-less random order in which it is established.

The explicit nature of the notation means one can switch between writing different parts of the specification without the danger of forgetting something. Because a response must be specified in every case, anything

which has not yet been specified is an instance of incompleteness. This applies right down to the level of the individual table cell. For example, we might initially define a rule for when the "open/close" button is pressed and the tray is closed. Because we are in a hurry to record something else, we fill only the cells for `Tray`, showing it changing from `closed` on the source row to `opening` on the target row; the other cells are left empty. When we later return to this rule, we need to fill in the blank cells for the `Mode` variable (and perhaps some more variables which have been added in the meantime). At this point, we realise that the tray doesn't always open on request. This is because we choose to regard recording as a critical operation which must be stopped explicitly before the tray can be opened. Therefore the press of the "open/close" button is ignored if `Mode` is `record`. We then complete the rule for those cases where `Mode` is not `record` (see rule `r1_1` in Table 2) and add another rule for when `Mode` is `record` (with a target row full of "no change" symbols).

4.4 Constraints Before Rules

Though the approach allows for the specification to be completed in a relatively random-access manner, a certain ordering between the sections is enforced since, e.g., one needs to define an event before one can define the response to that event in a rule. An ordering that is not naturally enforced but should still be followed is "constraints before rules." This means that, when new variables or values are defined, the cells in the corresponding columns of the constraint table should be filled before those in the transition table. Following this guideline ensures that the semantics of the variables have been properly thought about before they are used. More simply, it makes sense to define the possible states of the system before trying to define the transitions between them.

4.5 Communicating with the Stakeholder

Normally, the project stakeholder is not directly involved in producing the specification document and only reviews it when it has been completed. During the specification process then, the specifier meets the stakeholder frequently to discuss the required behaviour. The specification is written "off-line" and issues that are elicited are noted and discussed at the next meeting. There, the specifier typically communicates in a relatively informal way the issues that have arisen by translating the incomplete rules into verbal scenarios which demonstrate the behaviour in question, e.g.

> "Suppose the machine has started recording a track and, before the minimum track length has been recorded, a copy prohibit signal is detected, what should happen?"

The resolution is noted down in the meeting, the document is updated between meetings, and this cycle continues until the specification is complete.

4.6 Document Context and Use

The detailed specification which results from this approach is a reference document. It is not suitable to use as an introduction to the system. Usually, a separate informal summary of the system functionality will be written in natural language to provide a brief overview of the requirements. This may indicate, for example, that there is to be a "repeat" function in the CDR and perhaps the order in which the setting toggles, but omit details like its being reset when the tray is opened, etc.

The brief, informal specification is likely to be produced first and to serve as a starting point for the detailed specification. In such a document, ambiguities, inconsistencies and omissions are almost inevitable. It should be understood that where there is any doubt or conflict, the detailed specification takes precedence.

Depending on the development cycle and tools used, it may be convenient to use the specification as the basis for a set of test cases, with each test designed to exercise one or more of the transitions defined by a particular rule.

4.7 Overspecification

Occasionally, when deciding how the system should respond in a given case, there are several acceptable alternatives. With no compelling reason to choose one over another, an arbitrary choice may be made. However, a particular preference may emerge later, perhaps during implementation, and so the specification may need to be changed. For example, someone may notice that a rule is inconsistent with another rule in that it defines a different response in a similar case.

The earlier arbitrary choice may be called overspecification. After all, the need to change the specification might have been avoided if the choice had been left open. However, such an approach is not recommended. For one thing, it is much easier to work with a specification in which the response in each case is definite (deterministic) than one containing multiple alternatives for certain cases. Picking one alternative also makes it more likely that any problem with it will be found sooner. Further, leaving something open in a specification increases the danger that an unanticipated choice will be made during implementation – perhaps one that would not have been accepted in a review of the specification.

Though the method requires detail, overspecification has not been a problem in practice. In most cases, there is a definite requirement or at least a strong preference. When a situation like that described above arises,

it can be addressed by a simple change request. Agreement to the change is a formality since the proposed alternative would have been accepted originally in any case.

Therefore, when more than one response is equally acceptable, simply choose one and add a comment to the rule as a reminder that the choice is flexible in case a preference for an alternative later emerges.

4.8 Sparseness of Tables

A frequent initial observation is that the tables in the specification are quite sparse, with perhaps eight in every ten cells containing "don't care" or "no change" symbols. This is sometimes stated as an objection to the notation, on the grounds that it is inefficient. However, it is important to remember that the notation is designed to capture the desired behaviour in as direct and simple a manner as possible. To this end, every cell in the table serves a definite purpose, helping to ensure that nothing is overlooked. A "don't care" is not a wasted cell but an explicit record that a possible dependency, which needed to be considered, has in fact been considered. Thus, the notation is inefficient only with respect to representation and not with respect to elicitation or ease of reference, for which it is optimized. (Structuring, which would improve the efficiency of representation but distract from and disrupt the primary goals of the specification phase, may be addressed subsequently as part of a design phase. This is when it can best be done: When all the facts are already known and recorded in the specification, structuring decisions can be made with the full picture in mind.)

4.9 Redundancy in the Constraint Table

It may be noticed that there is redundancy in the constraint table. This is because the contents of the cells below the shaded diagonal could be derived from the contents of those above it, and vice versa. In theory therefore, a "triangular" table would be sufficient. The advantage of the full rectangular format table is that one may see the effect that a variable having a particular value has on all the other variables simply by reading across a row of the table.

A Formula Grammars

A.1 Constraint Formula Grammar

```
          Formula := [ FormulaSum [ "=>" Formula ] ]
       FormulaSum := FormulaProduct [ "||" FormulaSum ]
    FormulaProduct := FormulaUnary [ "&&" FormulaProduct ]
      FormulaUnary := [ "!" ] FormulaElement
   FormulaElement := ( "(" Formula ")" )
                   | ( variable-name "=" "{" CurrentValues "}" )
    CurrentValues := Values ( "|" Values )*
           Values := "*" | value-name | value-set
```

variable-name is an identifier, i.e., a letter or an underscore ('_') followed by any number of letters, digits or underscores

value-name is a sequence of one or more identifiers separated by '-'s

value-set is the same as *value-name* followed by "-*"

A.2 Rule Formula Grammar

The grammar for a rule formula is identical to that for a constraint formula except that the production for "FormulaElement" changes and another ("NewValues") is added:

```
          Formula := [ FormulaSum [ "=>" Formula ] ]
       FormulaSum := FormulaProduct [ "||" FormulaSum ]
    FormulaProduct := FormulaUnary [ "&&" FormulaProduct ]
      FormulaUnary := [ "!" ] FormulaElement
   FormulaElement := ( "(" Formula ")" ) | ( variable-name (
                     ( "=" "{" CurrentValues "}" [ AssignmentSequence ]
                     | AssignmentSequence ) )
AssignmentSequence := ( "->" "{" CurrentValues "}" )* "->" "{" NewValues "}"
    CurrentValues := Values ( "|" Values )*
       NewValues := "=" | ( Values ( "|" Values )* )
           Values := "*" | value-name | value-set
```

Note that the use of "=>" in a rule formula is strongly discouraged (see Section 3.6).

A.3 Comments

In constraint and rule formulas, as in several scripting languages, a '#' character indicates that everything up to the end of the same line is a comment, that is, text which is not to be considered part of the formula.

```
# Formula with comments.
Alpha = {a1 | a3} # Alpha cannot be a2
&& !Beta = {b2} # nor can Beta have
# the value b2
&& Gamma = {g4}
```